UNIVERSITÀ DEGLI STUDI DI PISA

Facoltà di Scienze Matematiche, Fisiche e Naturali
Corso di Laurea in Scienze dell'Informazione

Tesi di Laurea

Progetto ed implementazione di una modifica al compilatore IDL di CORBA per produrre codice che consente di utilizzare oggetti CORBA come componenti COM

Candidato
Massimo Di Giorgio

Relatore
Prof. Giuseppe Attardi

Controrelatore
Prof. Gian Luigi Ferrari

Anno Accademico 1998/99

Ai miei genitori,
a mia zia Angelina
ed a mio zio Antonio

Indice

Introduzione

Nel corso degli anni sono state sviluppate diverse tecniche e metodologie per affrontare lo spinoso problema dello sviluppo del software: programmazione strutturata, modularizzazione, strutture dati astratte, programmazione ad oggetti.

La programmazione ad oggetti incorpora i principi di astrazione, modularizzazione, incapsulamento ed ereditarietá. Quest'ultima costituisce il principale meccanismo per il riutilizzo di codice preesistente.

Il modello ad oggetti si è evoluto recentemente verso la creazione di **Framework** (librerie di classi astratte) o di **Software Patterns**. Tuttavia le soluzioni basate su un linguaggio di programmazione ad oggetti restano limitate perché:

- richiedono che tutto il codice sia scritto nello stesso linguaggio di programmazione,
- richiedono compatibilitá binaria,
- non supportano applicazioni destribuite,
- non forniscono un ambiente di composizione.

Se è vero che il mercato produce ció che il mercato richiede, non c'è da stupirsi se l'esigenza di suddividere un'applicazione in piú parti e la possibilitá di poter riutilizzare codice giá scritto, hanno portato alla nascita dei componenti software o **Software Components**.

L'obbiettivo della tesi è la progettazione e lo sviluppo di uno strumento per integrare fra di loro i componenti software sviluppati in COM e CORBA, le principali architetture ad oggetti presenti attualmente sul mercato.

Nel primo capitolo spieghiamo cosa sono i componenti software, quali sono i loro principali requisiti e quali sono i loro principali vantaggi. Non manca una breve esposizione su cosa offre oggi il mercato.

Il secondo capitolo spiega, entrando nei particolari, il **Component Object Module** o COM di Microsoft. Partendo da zero si descrivono tutti i passi necessari per creare un componente senza nascondere i dettagli dell'implementazione che gli ultimi **tool** di sviluppo della Microsoft celano abilmente. Anche il lato client viene trattato abbondantemente presentando degli esempi sia in C++, sia in Java e sia in Visual Basic.

La **Common Object Request Broker Architecture** o CORBA è invece trattata nel terzo capitolo. Dopo una carrellata sulla sua struttura, si pone particolare attenzione sul confronto tra invocazioni statiche e invocazioni dinamiche. Anche qui si crea un componente completo anche se, siccome esistono varie distribuzioni di CORBA, gli esempi di codice non sono universali ma sono legati alla particolare implementazione fornita da Mico.

Il quarto capitolo descrive la specifica CORBA relativa all'interazione tra COM e CORBA. Si mettono in evidenza le caratteristiche comuni dei due sistemi e, soprattutto, le loro differenze. Viene data inoltre una visione generale di come potrebbe essere strutturata una possibile architettura di integrazione.

CORBA descrive i suoi componenti con l'**OMG IDL** (**Object Menagement Group Interface Definition Language**), mentre COM usa il **MIDL** (**Microsoft Interface Definition Language**). Il primo passo per avvicinare i due sistemi è quello di rendere compatibili i due linguaggi. Nel quinto capitolo si entra nel vivo dell'implementazione, descrivendo proprio come mappare l'OMG IDL sul Microsoft IDL secondo la specifica CORBA.

Il sesto capitolo presenta l'estensione realizzata del compilatore IDL di Mico per fargli produrre codice MIDL. Si descrive brevemente lo schema generale del traduttore, le strutture dati che usa, i suoi pregi ed i suoi difetti. Particolare attenzione è stata posta ai punti in cui non si è rispettata la specifica CORBA, ma è stata proposta un'implementazione particolare. Il lavoro è stato apprezzato dagli stessi autori di Mico, che lo hanno incluso nella distribuzione ufficiale (per maggiori approfondimenti consultare il sito *http://www.mico.org*).

Seguono infine le conclusioni su questo (mio malgrado lungo) lavoro, mettendo in evidenza le difficoltá incontrate e i possibili miglioramenti.

Capitolo 1

Introduzione ai componenti

In questa breve introduzione vedremo cosa sono i componenti software (**Software Components**), quali sono i loro vantaggi, perché sono nati e quali sono attualmente le principali architetture che li supportano.

1.1 I componenti

Solitamente un'applicazione è costituita da un singolo e monolitico file binario. Una volta che il compilatore ha generato l'applicazione, questa non cambia fino a quando non viene ricompilata e pubblicata la versione successiva. Eventuali cambiamenti nei sistemi operativi, nel tipo di hardware e nelle necessità del cliente devono attendere la ricompilazione dell'intera applicazione. Con l'attuale ritmo di cambiamento nell'industria del software, le applicazioni non possono permettersi il lusso di rimanere statiche dopo la loro pubblicazione. Gli sviluppatori devono trovare il modo di dare nuova vita alle applicazioni che hanno già prodotto. La soluzione è quella di suddividere l'applicazione monolitica in pezzi separati, in altre parole componenti (Figura 1.1).

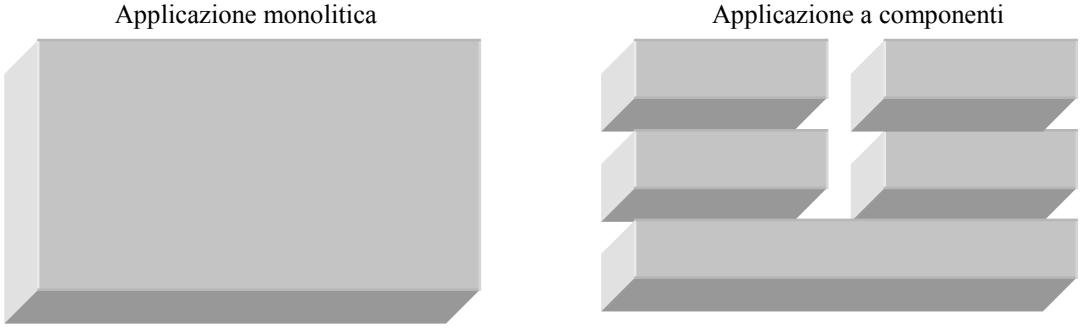


Figura 1.1 Suddividere un'applicazione monolitica (sinistra) in componenti (destra) la rende adattabile ai cambiamenti.

Con l'avanzamento della tecnologia, nuovi componenti possono rimpiazzare quelli esistenti che formano l'applicazione. Questa non è più un'entità statica, destinata a

diventare obsoleta ancora prima di essere pubblicata. Al contrario, l'applicazione si evolve con grazia nel tempo, man mano che i nuovi componenti vanno a sostituire i più vecchi. Possono essere rapidamente costruite applicazioni totalmente nuove da componenti già esistenti.

Tradizionalmente, ogni applicazione viene suddivisa in file, moduli o classi, che sono compilati e *linkati* in modo da formare l'applicazione monolitica. Costruire applicazioni con componenti, cioè il processo conosciuto come l'*architettura basata su componenti* è tutt'altra cosa. Un componente è come una mini applicazione: viene fornito come pacchetto di **codice binario compilato**, *linkato* e pronto per l'uso. L'applicazione monolitica non esiste più. Al suo posto troviamo un componente personalizzato, che durante l'esecuzione si connette ad altri componenti, per formare un'applicazione. Modificare o migliorare l'applicazione è puramente una questione di sostituire uno dei componenti costitutivi con uno di nuova versione. Se si vuole aggiungere un'ulteriore funzionalità all'applicazione o semplicemente si vuole migliorare un particolare algoritmo, non è più necessario ricostruire tutta l'applicazione, ma basta sostituire un componente con uno più recente che implementa le nuove funzionalità (Figura 1.2).

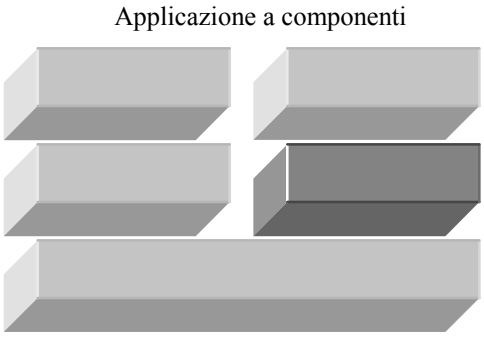


Figura 1.2 Un componente nuovo (più scuro) e migliorato ha sostituito il vecchio componente.

Aumentando l'ampiezza di banda e l'importanza delle reti, non potrà che aumentare la necessità di applicazioni composte da parti sparse in ogni punto di una rete.

L'architettura basata su componenti aiuta a semplificare il processo dello sviluppo di tali applicazioni distribuite. Le applicazioni client/server hanno già compiuto i primi passi verso un'architettura basata su componenti, suddividendosi in due parti: quella client (che richiama e usa i componenti) e quella server (che li implementa).

Costruire un'applicazione distribuita sulla base di un'applicazione esistente è più facile se quest'ultima è costituita a componenti. Per prima cosa, l'applicazione è già stata suddivisa in parti funzionali che possono trovare una collocazione remota. In secondo luogo, poiché i componenti sono sostituibili, si può sostituire un componente con un altro che abbia il preciso scopo di comunicare con un componente remoto. Per esempio, nella Figura 1.3, due componenti sono stati collocati su diverse macchine remote della rete. Sulla macchina locale, essi sono

stati sostituiti da due nuovi componenti (più scuri) che inviano, attraverso la rete, richieste provenienti dagli altri componenti (Figura 1.3).

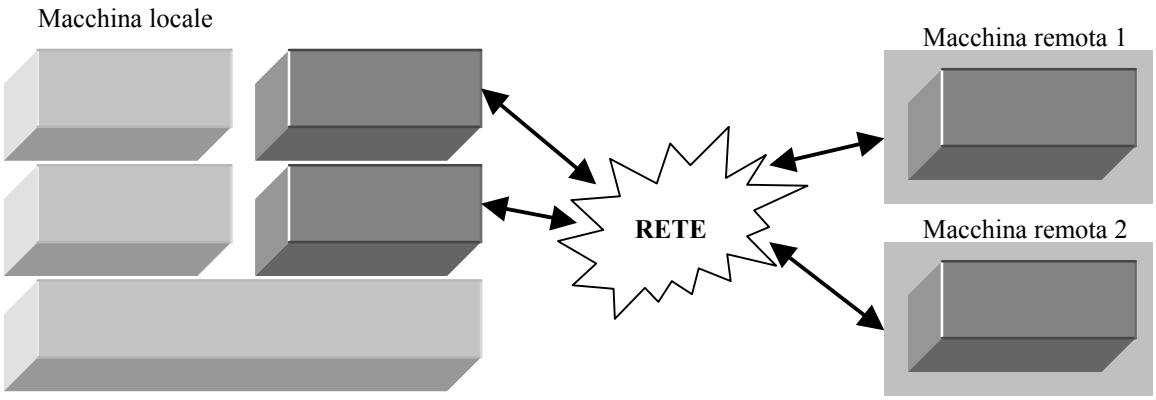


Figura 1.3 Componenti situati su un sistema remoto in rete.

All'applicazione sulla macchina locale non interessa che gli effettivi componenti siano in postazione remota. Allo stesso modo, ai componenti remoti non interessa di essere tali. Con l'apposito componente per la comunicazione in remoto, l'applicazione può totalmente ignorare dove gli altri componenti siano effettivamente situati.

1.2 Requisiti dei componenti

I vantaggi dell'utilizzo dei componenti sono direttamente correlati alla loro capacità di connettersi e disconnettersi in modo dinamico da un'applicazione. Al fine di garantirsi tale capacità, i componenti devono soddisfare due requisiti. Il primo è che i componenti devono collegarsi fra di loro in modo dinamico, il secondo è che i componenti devono nascondere (o *incapsulare*) i dettagli relativi al modo in cui sono stati implementati. Ognuno dei due requisiti dipende dall'altro. Il collegamento dinamico è un requisito essenziale di un componente, mentre il fatto di celare le informazioni è condizione necessaria per il collegamento dinamico. Vediamo in dettaglio i due requisiti:

- **Collegamento dinamico**
  Il fine ultimo è quello di dare all'utente finale la capacità di sostituire i componenti dell'applicazione mentre esegue l'applicazione stessa. Il supporto per la modifica dei componenti in fase di esecuzione richiede la capacità di collegare i componenti tra loro in modo dinamico.

- **Incapsulamento**
  Per comprendere l'incapsulamento abbiamo bisogno di stabilire il significato di alcuni termini che saranno spiegati in dettaglio successivamente. Un programma o un componente che utilizza un altro componente si chiama **client**. Un client è collegato a un componente attraverso un'**interfaccia**. Se il componente cambia senza modificare l'interfaccia, il client non necessariamente deve essere modificato. Allo stesso modo, se il client cambia senza modifiche all'interfaccia, il componente non deve essere cambiato. Tuttavia se la modifica o del componente o del client implica la modifica dell'interfaccia, anche l'altro lato dell'interfaccia deve essere modificato.

  Pertanto, per poter usufruire del collegamento dinamico, i componenti e i client devono fare di tutto per non modificare le proprie interfacce. Essi devono essere incapsulati. I dettagli su come il client e il componente sono implementati non devono essere evidenti nell'interfaccia. Più l'interfaccia è tenuta separata dai dettagli di implementazione, meno probabile è che l'interfaccia possa essere modificata in conseguenza di una modifica del client o del componente. Se l'interfaccia non cambia, modificare un componente avrà un effetto limitato sulla parte restante dell'applicazione.
  Isolare il client dall'implementazione del componente implica alcuni importanti vincoli per il componente. Questi vincoli sono i seguenti:

  1. Il componente non deve mostrare il linguaggio di programmazione utilizzato per la sua implementazione. Ogni client dovrebbe essere in grado di utilizzare qualsiasi componente a prescindere dal linguaggio con il quale il client o il componente sono stati scritti.
  2. I componenti devono essere forniti in formato binario. Se i componenti devono poter nascondere il loro linguaggio di implementazione, devono essere forniti già compilati, *linkati* e pronti all'uso.
  3. I componenti devono poter essere aggiornati senza impatto per gli utilizzatori correnti. Nuove versioni di un componente dovrebbero lavorare sia con i vecchi, sia con i nuovi client.
  4. I componenti devono essere riposizionabili in modo trasparente su una rete. Un componente e il programma che lo utilizza devono poter essere eseguiti nello stesso processo, in processi diversi, o su macchine diverse. Il client dovrebbe essere in grado di trattare un componente remoto allo stesso modo di un componente locale. Se i componenti remoti fossero trattati in modo diverso da quelli locali, il client dovrebbe essere ricompilato ogni qualvolta un componente locale è spostato altrove nella rete.

1.3 Cosa offre il mercato

Ci sono altre forme per utilizzare oggetti già compilati? Per un programmatore Windows la risposta è semplice: una DLL (un modulo per altri Sistemi Operativi). Ma quali sono i problemi di una DLL? Per prima cosa le DLL non sono

necessariamente indipendenti dal linguaggio di programmazione. Basti pensare alla convenzione di chiamata di funzione (i parametri passati sullo stack, il loro ordine e la loro eliminazione), i problemi di decorazione dei nomi del C++, ecc. Supponiamo comunque di aver risolto questi problemi e di essere riusciti a sviluppare una DLL funzionante. Dopo un po' di tempo (nell'informatica il tempo è molto relativo) arriva il momento di aggiornare il nostro componente. Se aggiungiamo delle nuove funzioni virtuali alla fine del nostro oggetto la cosa potrebbe funzionare. Ma non è così. Abbiamo, infatti, *shiftato* i puntatori delle funzioni virtuali di tutti quegli oggetti che ereditano dal nostro. Siccome una chiamata ad una funzione virtuale ha bisogno di un *offset* fisso dentro la *vtable* per chiamare la funzione corretta, non posso aggiungere funzioni virtuali perché altrimenti modificherei la *vtable* (lo posso fare a patto di ricompilare ogni programma che usa il nostro oggetto o qualsiasi altro oggetto derivato da lui. Ancora, se il client usa *new* per allocare gli oggetti, non possiamo cambiare la dimensione del nostro componente se non ricompilando di nuovo tutto. Per ovviare a questo problema si potrebbe sviluppare una nuova DLL con un nuovo nome. Ma per far funzionare i vecchi client dovrò conservare anche la vecchia determinando così uno spreco di memoria e di spazio sul disco fisso (si provi a guardare quante DLL simili sono presenti nella directory "**system"** di Windows).

Un'altra alternativa è la programmazione diretta dei **socket**. Nei sistemi moderni, la comunicazione tra macchine, e a volte tra processi della stessa macchina, è realizzata tramite l'uso dei socket. Semplicemente parlando, un socket è un canale grazie al quale le applicazioni si possono connettere e comunicare fra di loro. Il metodo più potente per far comunicare i componenti di un'applicazione è quindi l'uso diretto dei socket (**socket programming**), che significa che il programmatore deve leggere e scrivere i dati direttamente dal socket.

Le **Application Programming Interface** o **API** per la programmazione dei socket sono a basso livello. Quindi, l'*overhead* che si introduce in ogni applicazione è molto basso. Dall'altra faccia della medaglia, però, le API dei socket non permettono la gestione diretta dei tipi di dato complessi, specialmente quando i componenti risiedono su macchine diverse, o sono implementati in linguaggi di programmazione differenti.

In conclusione, la programmazione dei socket è più adatta per la creazione di piccole applicazioni e non di programmi complessi che risiederanno su macchine diverse.

Ad un livello superiore della programmazione dei socket, troviamo le **Remote Procedure Call** o **RPC**. Le RPC forniscono un'interfaccia ad alto livello per la programmazione dei socket. Usando le RPC il programmatore definisce delle funzioni (simili a quelle di un linguaggio funzionale) e genera del codice che permette di usarle trasparentemente come se fossero delle normali funzioni. Il chiamante non si accorge della differenza. Tutta la procedura dell'invio e della ricezione dei dati tramite il socket è gestita automaticamente.

Tuttavia non c'è uno standard unico per le RPC, ma ci sono varie versioni che sono specifiche del loro produttore. Inoltre le RPC rimangono ancora a basso livello e non forniscono nessun supporto per la localizzazione dei componenti, la gestione delle versioni, ecc.

Un'ulteriore alternativa, che è molto di moda in questo momento, potrebbe essere l'uso del linguaggio Java. Ad esempio si potrebbero usare i **Java Beans**, ma questi lavorano solo con programmi scritti in Java (non è vero del tutto dato che la *virtual machine* di Microsoft permette di usare i Java Beans come oggetti COM, e Sun ha sviluppato un "ponte" Java/ActiveX); ma in generale, Java è un sistema a linguaggio singolo: i componenti Java possono essere usati solo in programmi Java. Un altro problema che nasce usando Java è quello di dover decidere, quando si scrive un programma, dove sarà locato il componente (locale o remoto) e di adattare il programma in base a questa scelta. Infine Java non permette di destreggiarsi con le varie versioni dei componenti, ed è molto più lento di altri linguaggi come ad esempio il C++ (per rendersi conto di ciò si possono confrontare i due linguaggi nella manipolazione di stringhe e array, due operazioni dove le prestazioni sono molto differenti).

Un'altra possibilità potrebbe essere quella di usare i **Java Remote Method Invocation** o **RMI**, un'architettura molto simile a COM e CORBA. Un vantaggio dei RMI è che supportano il passaggio degli oggetti per valore (una caratteristica che è stata aggiunta solo ultimamente in CORBA e che non è supportata in COM), mentre uno svantaggio è che i RMI funzionano solo con Java, cioè sia il client che il server devono essere scritti in Java.

1.4 CORBA e COM

Abbiamo visto che un componente per essere definito tale deve avere alcuni requisiti e deve rispettare alcune regole. Quindi tutti possono scrivere dei componenti, è sufficiente trovare il sistema per rispettare questi vincoli. C'è solo un piccolo problema: se invento un modo per sviluppare dei componenti, solo quelli che conoscono questo modo possono a loro volta sviluppare client e componenti compatibili con i miei. Bisogna allora non solo inventare un metodo, ma anche renderlo un standard in modo tale che ogni sviluppatore sia in grado di produrre componenti compatibili con questo standard.

CORBA di OMG e **COM** di Microsoft non sono altro che delle **specifiche** per sviluppare dei componenti che rispettino i vincoli visti prima (ed altri aggiunti in base alla specifica). Ma per poter sviluppare realmente dei componenti, oltre a queste specifiche servono anche degli strumenti che si occupano della comunicazione tra client e **server** (il programma che implementa i componenti) e della traduzione dei parametri o **marshaling.** Ancora una volta chiunque può sviluppare una sua versione di questi strumenti (sempre attenendosi alle specifiche di CORBA e COM).

In questo testo si prenderanno in considerazione le implementazioni di Microsoft per COM e **Mico** per CORBA. Mico è l'acronimo di **M**ico **I**s **CO**RBA. È un progetto che ha l'intenzione di fornire un'implementazione completa e gratuita dello standard CORBA. La differenza con le altre implementazioni è che Mico è stato sviluppato per scopo educativo e tutti i sorgenti sono disponibili tramite licenza GNU.

Capitolo 2

COM

In questo capitolo descriveremo l'architettura **COM** di Microsoft. Spiegheremo le sue caratteristiche principali, i suoi vantaggi e i suoi limiti. Tutti gli esempi di codice saranno scritti in C++ (se non diversamente specificato) perché è il suo linguaggio nativo.

2.1 Il Component Object Module

COM è l'acronimo di **Component Object Module** ed è la proposta di Microsoft per sviluppare ed utilizzare componenti. COM è soprattutto una specifica. Specifica in che modo vadano costruiti componenti che possono essere dinamicamente intercambiabili. COM fornisce lo standard che i componenti ed i client seguono per garantire di poter lavorare insieme. Gli standard sono altrettanto importanti per le architetture basate su componenti, quanto lo sono per qualsiasi sistema dotato di parti intercambiabili. Lo standard COM è contenuto nella **COM Specification** [COM Spe], distribuito da Microsoft.

2.2 I componenti COM sono…

I componenti COM sono costituiti da un codice eseguibile distribuito sotto forma di DLL, o sotto forma di file eseguibile (.EXE), entrambi per Win32 (per Win16 c'è solo la tecnologia **OLE Automation**). I componenti scritti in base allo standard COM soddisfano tutti i requisiti di un'architettura basata su componenti.
I componenti COM si collegano in modo dinamico. COM utilizza DLL per collegare i componenti in modo dinamico, ma come abbiamo visto, il collegamento dinamico di per sé non costituisce una garanzia per un'architettura basata su componenti. I componenti devono essere anche incapsulati. I componenti COM

possono essere incapsulati facilmente perché rispettano i vincoli che abbiamo visto prima:

- sono completamente indipendenti dal linguaggio. Possono essere sviluppati utilizzando praticamente qualsiasi linguaggio procedurale da **Ada** a **C** a **Java** a **Modula-3** a **Pascal**. Qualsiasi linguaggio può essere modificato in modo da poter utilizzare componenti COM, incluso il **Visual Basic**. Infatti, esistono diversi modi per scrivere componenti COM che possono essere impiegati da linguaggi macro. Comunque il linguaggio nativo di COM è il **C++** e la maggior parte degli esempi in questo documento saranno scritti con questo linguaggio.
- possono essere forniti in formato binario.
- possono essere aggiornati senza compromettere i client esistenti.
- Possono essere riposizionati in modo trasparente su una rete. Un componente su un sistema remoto è trattato allo stesso modo di un componente su sistema locale perché dichiarano la loro presenza in modo standardizzato. Utilizzando lo schema di pubblicazione COM, i client possono trovare in modo dinamico i componenti di cui hanno bisogno.

COM è di più della sola specifica. In effetti, ha una certa dose d'implementazione. Esso ha un'**API**, la libreria COM, la quale offre servizi di gestione dei componenti che sono utili per tutti i client e tutti i componenti. La maggior parte delle funzioni dell'**API** non è particolarmente difficile da implementare direttamente quando si sviluppano componenti sullo stile COM in un sistema non Windows. La libreria COM è stata scritta in modo da garantire che le operazioni principali vengano svolte allo stesso modo da tutti i componenti. Questa libreria, inoltre, fa risparmiare tempo agli sviluppatori durante l'implementazione dei propri componenti e dei client. Gran parte del codice di libreria COM è di supporto per componenti distribuiti, o in rete. L'implementazione di COM (in questo caso si parla di **DCOM** ovvero **Distributed COM**) in sistemi Windows, fornisce il codice che serve per comunicare con i componenti presenti in una rete. Questo non soltanto permette di evitare di dover scrivere il codice per la messa in rete, ma anche di dover sapere come scriverlo.

2.2.1 Tipi di server

- **InProcServer**: in questo caso il server è costituito da una DLL che viene caricata nello stesso spazio logico del client. È molto efficiente in termini di velocità (costa come la chiamata di una funzione virtuale C++).
- **OutOfProcServer**: il server è un programma (.EXE) che gira sulla stessa macchina del client. Esistono molti modi per comunicare tra processi.COM utilizza le **Local Procedure Call** o **LPC** che si basano sulle **Remote Procedure Call** o **RPC**. Ma queste non bastano; infatti, si devono anche ottenere i parametri passati ad una funzione dallo spazio d'indirizzamento del client a quello del componente (server). Questa traduzione dei parametri si chiama **marshaling**. È possibile creare un **marshaling** proprietario implementando l'interfaccia **IMarshal**, ma la libreria COM ne implementa una versione standard che funziona praticamente per qualsiasi interfaccia. Il funzionamento

generale è questo: il client comunica con una DLL (chiamata **proxy**); il **proxy** effettua il **marshaling** dei parametri delle funzioni e richiama un'altra DLL (chiamata **stub**) tramite le **LPC**. La DLL **stub** effettua il **de-marshaling** dei parametri e richiama la corretta funzione dell'interfaccia nel componente, passandogli i parametri (Figura 2.1).

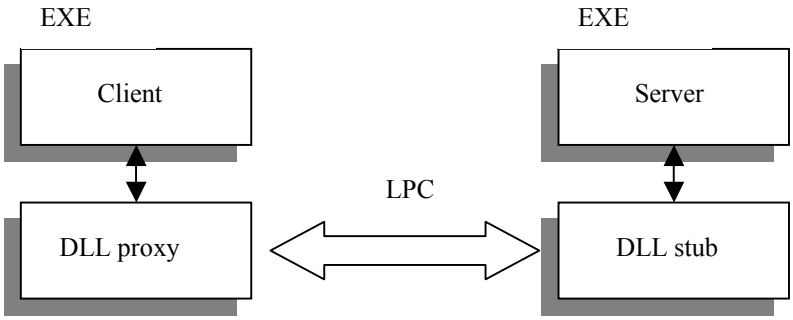
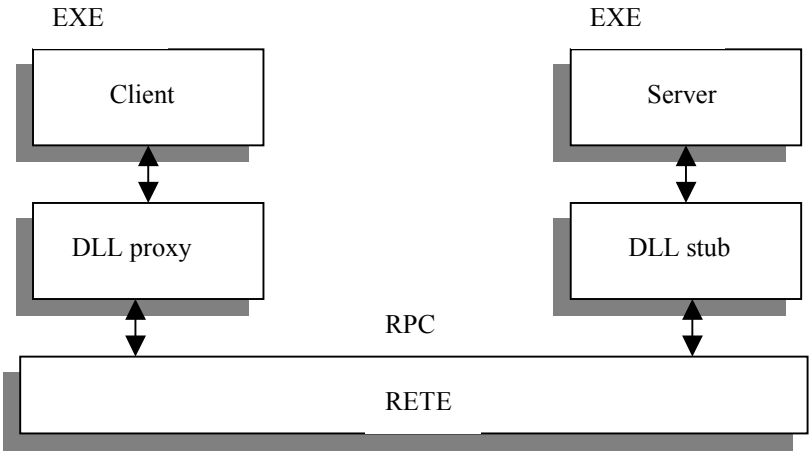


Figura 2.1 Schema di funzionamento di un componente **OutOfProcServer**.

- **RemoteServer**: il server è una DLL o un programma (.EXE) che si trovano su una macchina diversa da quella del client. Lo schema di funzionamento è simile a quello precedente, solo che si usano le **RPC** invece delle **LPC** e il **marshaling** diventa più complicato. Infatti, i parametri, oltre ad essere tradotti, devono essere trasformati in pacchetti e spediti sulla rete che collega le due macchine (Figura 2.2).



2.2.2 Oggetti

2.2.3 Interfacce

Un componente, per essere utile, deve poter comunicare con il mondo esterno; è qui che intervengono le interfacce. La sola maniera per accedere ad un oggetto COM è attraverso un'interfaccia. Un'interfaccia è due cose.

Per prima cosa è un insieme di funzioni che si possono chiamare per interagire con il componente. In C++ le interfacce sono rappresentate come **classi di base astratte**. Per esempio la definizione di un'interfaccia **IFoo** potrebbe essere:

```
class IFoo {
     virtual void Func1(void) = 0;
     virtual void Func2(int nCount) = 0;
};
```

Ignoriamo per il momento il tipo di ritorno e l'ereditarietà, ma notiamo che ci possono essere più funzioni in un'interfaccia e che tutte le funzioni sono **virtuali pure**: non hanno un'implementazione nella classe **IFoo.** Non stiamo definendo il comportamento qui, stiamo solo definendo quali funzioni ci sono nell'interfaccia (un oggetto reale deve avere un'implementazione naturalmente).

Per seconda cosa, e più importante, un'interfaccia è un *contratto* tra il componente e il suo client. In altre parole, un'interfaccia non solo definisce quali funzioni sono disponibili, ma definisce anche cosa l'oggetto fa' quando una funzione è chiamata. Questa definizione semantica non è in termini di specifica implementazione dell'oggetto, per questo motivo non c'è modo di rappresentarla in codice C++ (potremmo però implementare il componente in C++). Invece, la definizione è in termini del comportamento dell'oggetto, così delle eventuali revisioni e/o nuovi oggetti che implementano la stessa interfaccia, sono possibili. Infatti, l'oggetto è libero di implementare il *contratto* in qualsiasi modo. In altre parole il fornitore del componente, per permettere di utilizzarlo, deve documentare le funzioni che è possibile chiamare e cosa fanno queste funzioni **non nel codice sorgente**. Questo è particolarmente importante perché i client non hanno il codice sorgente del componente.

In COM, una volta che il componente è stato sviluppato e venduto, l'interfaccia non si può più cambiare. Non si può né aggiungere, né cancellare, né modificare niente. Questo perché altri componenti possono dipendere dal nostro. Si può cambiare l'implementazione, ma l'interfaccia deve rimanere la stessa. Se devo aggiungere delle nuove potenzialità al componente **devo scrivere una nuova interfaccia** e implementarla nel componente. In questo modo funzioneranno sia i vecchi client sia i nuovi (i quali potranno giovarsi delle nuove potenzialità aggiunte). La stessa Microsoft ha fatto così; infatti, dando uno sguardo alle funzioni di libreria si possono vedere casi del genere (**IClassFactory** e **IClassFactory2**, **IViewObject** e **IViewObject2**). Quindi per aggiungere nuove funzionalità non ci resta che scrivere un interfaccia **IFoo2**.

2.2.4 Interfacce multiple

In COM un oggetto può supportare molte interfacce. Infatti, tutti gli oggetti COM supportano almeno due interfacce (almeno l'interfaccia **IUnknown**, che vedremo a cosa serve, e l'interfaccia che fa fare al componente quello che noi vogliamo). I controlli **ActiveX** (che si basano su COM) supportano almeno una dozzina di interfacce, la maggior parte delle quali sono standard, cioè uguali per tutti.
Per permettere ad un componente di supportare un'interfaccia, dobbiamo implementare ogni funzione dell'interfaccia. Non è un lavoro da niente soprattutto nel caso degli **ActiveX**. Per questo motivo la Microsoft ha sviluppato dei tool e delle classi che semplificano quest'operazione: la **MFC** (**Microsoft Foundation Class**) e soprattutto l'**ATL** (**Active Template Library**).

2.2.5 Ereditarietà

Torniamo al discorso di aggiungere potenzialità al nostro componente. È un lavoro inutile creare una nuova interfaccia che semplicemente estende una già esistente. Per questo motivo COM supporta l'**ereditarietà** sulle interfacce (a differenza di CORBA, COM supporta solo l'ereditarietà singola sulle interfacce e non supporta l'ereditarietà sulle implementazioni; fornisce però dei metodi alternativi che sono il **contenimento** e l'**aggregazione**. Anche di questo parleremo in seguito). Quindi potrei definire la nuova interfaccia **IFoo2** nel seguente modo:

```
class IFoo2 : public IFoo {
     // Eredita Func1 e Func2
     virtual void Func2Ex(double nCount) = 0;
};
```

In questo modo l'interfaccia **IFoo2** eredita le funzioni **Func1 e Func2** da **IFoo**. Se implemento **IFoo2** dovrò comunque implementare queste due funzioni.

2.3 Chiamare i metodi dell'interfaccia

I metodi COM si chiamano come una comune funzione virtuale del C++, non si deve far altro che ottenere un puntatore ad un oggetto che implementa un'interfaccia e chiamare il metodo con questo puntatore.
Assumiamo di avere una classe C++ chiamata **CFoo** che implementa l'interfaccia **IFoo**:

```
class CFoo : public IFoo {
     void Func1() { /* . . . */}
     void Func2(int nCount) { /* . . . */)
};
```

È da notare che ereditiamo da **IFoo** per assicurarci di implementare la giusta interfaccia, nell'ordine giusto (l'ordine delle funzioni di interfaccia è importante).
Se otteniamo (in qualche modo) il puntatore all'interfaccia, i metodi si possono chiamare con un codice simile a questo:

```
#include <IFoo.h> // Non abbiamo bisogno di CFoo, ma solo di
IFoo

void DoFoo() {

     IFoo *pFoo = . . . // sarà spiegato in seguito

     // chiamate ai metodi
     pFoo -> Func1();
     pFoo -> Func2(5);
};
```

Dall'esterno è molto semplice, ma ci sono molte cose che accadono e che non riusciamo a vedere:

1. **pFoo** è dereferenziato per cercare la **vtable** dell'oggetto.
2. con un **offset** si punta alla funzione da chiamare.
3. la funzione viene chiamata.

In C++ ogni volta che si ha una funzione virtuale si ha anche una **vtable** che punta a questa funzione e la chiamata è fatta sempre con lo stesso meccanismo. Quindi la specifica COM e la chiamata di una funzione virtuale del C++ **coincidono** (sarà un caso?).
La cosa importante è, come abbiamo già detto, che per poter sviluppare oggetti COM in un qualsiasi linguaggio (almeno tutti quelli che supportano array di puntatori a funzioni) e quindi indipendentemente dal C++ era necessario uno standard per la chiamata delle funzioni membro di un componente (almeno per i linguaggi diversi dal C++). Comunque COM non si limita a questo. È molto difficile far comunicare due moduli binari, soprattutto se si trovano su due processi diversi o peggio ancora su macchine diverse, se usano Sistemi Operativi diversi. COM rende tutto questo trasparente ai programmatori in modo tale che questi si possono concentrare esclusivamente allo sviluppo dei componenti, senza preoccuparsi dei dettagli.

In Figura 2.3 è rappresentato lo schema di chiamata delle funzioni virtuali tramite il puntatore all'interfaccia.

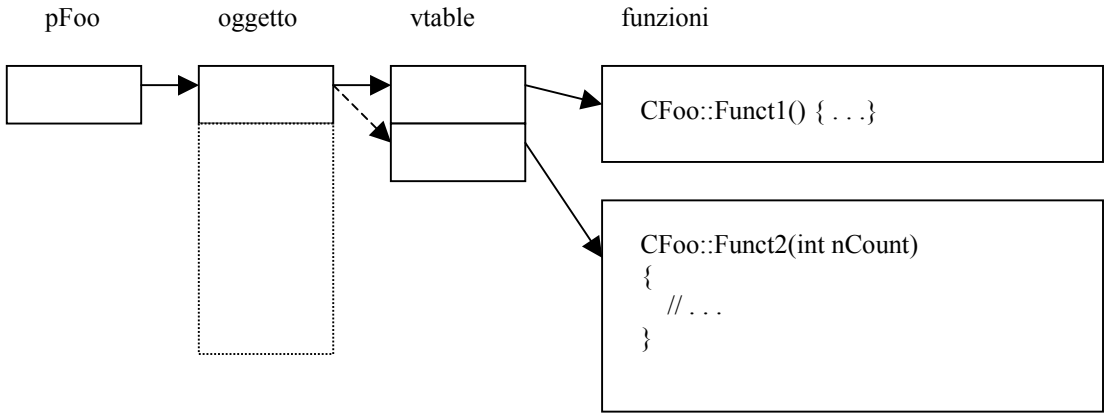


Figura 2.3 Chiamata di funzione virtuale tramite puntatore a interfaccia.

2.4 Identificatori in COM

Abbiamo bisogno di identificatori per poter indicare univocamente le interfacce, ma anche gli oggetti (perché non c'è una corrispondenza uno a uno tra interfacce e oggetti che le implementano). Potremmo usare degli interi a 32 bit, oppure a 64 bit, ma c'è un problema. Gli identificatori devono essere unici su tutte le macchine, poiché non c'è modo di sapere su quali macchine i componenti verranno installati.

Gli oggetti e le interfacce hanno bisogno degli stessi identificatori su tutte le macchine, così qualsiasi client può usarli. Inoltre nessun altro oggetto o interfaccia deve usare lo stesso identificatore di un altro. In altre parole questi identificatori devono essere **globalmente unici**.

Fortunatamente, gli algoritmi e le strutture dati esistono per creare questi identificatori. Usando la **network card ID** unica di ogni macchina, il tempo corrente, ed altre strutture dati, questi identificatori, chiamati **GUID** (globally unique identifiers), possono essere generati da un programma chiamato **GUIDGEN.EXE**, oppure chiamando la funzione **CoCreateGuid** delle API di Win32.I **GUID** sono memorizzati in una struttura di 16 byte (128 bit), dando la possibilità così di poterne generare 2^{128} diversi.

In C++ ci sono delle strutture dati definite negli header file di COM per i **GUID**, **CLSID** (identificatore per le classi), e **IID** (identificatore per le interfacce). Poiché queste strutture di 16 byte sono pesanti se vengono passate per valore, si usano allora i **reference** a queste strutture (**REFCLSID** e **REFIID**) quando si passano dei **GUID**.

È necessario creare un **CLSID** per ogni oggetto e un **IID** per ogni interfaccia che si vuole creare. COM definisce un insieme di interfacce standard e i loro **IID** associati. Per esempio la madre di tutte le interfacce, **IUnknown**, ha il suo unico **IID** che è "00000000-0000-0000-c000-000000000046". È un numero abbastanza complicato da gestire, ma per fortuna ci si riferirà ad un **GUID** sempre tramite il suo nome associato (in questo caso è **IID_IUnknown**).

2.4.1 GUID e il registro di Windows

Abbiamo visto che per ogni interfaccia e per ogni oggetto c'è un **GUID**. Ma come è creata quest'associazione? La risposta è semplice, nel registro di Windows. Quindi, una volta che abbiamo creato ad esempio la classe **CFoo** che implementa l'interfaccia **IFoo** dobbiamo registrare il componente. Tutti i tool di sviluppo che supportano COM lo fanno automaticamente, ma è possibile farlo anche *a mano*. Usando ad esempio il programma **REGEDIT.EXE** si deve aggiungere:

```
HKEY_CLASSES_ROOT

  CLSID
     {XXXXXXXX-XXXX-XXXX-XXXXXXXXXXXX} = Foo Class
          InProcServer32=c:\\Esempi\Foo\\Debug\\Foo.dll

HKEY_CLASSES_ROOT
  Interface
     {XXXXXXXX-XXXX-XXXX-XXXXXXXXXXXX} = IFoo
```

Le X indicano i due diversi **GUID** che si devono generare e **InProcServer32** indica il tipo di server per il componente (una DLL) e il *path* di dove si trova.
Molti oggetti hanno alcune voci addizionali, tipo la versione, il proxy, la **TypeLib**, ma li possiamo per il momento ignorare.
È possibile per un modulo (DLL o EXE) implementare più di un oggetto COM. Quando questo accade, ci sono più voci di **CLSID** che si riferiscono allo stesso modulo. Quindi si può definire la relazione tra moduli, classi e interfacce. Un modulo (l'unità di base che si crea e s'installa) può implementare uno o più componenti. Ogni componente ha il suo proprio **CLSID** e una voce nel registro che punta al nome del file del modulo. E ogni componente implementa almeno due interfacce: **IUnknown** e l'interfaccia che espone le funzionalità del componente.

2.5 Interfacce custom

Le interfacce custom sono quelle create da un programmatore, in altre parole sono le interfacce che non fanno parte delle interfacce standard di COM. Dobbiamo creare un **IID** per ogni nuova interfaccia e definire le funzioni in essa contenute.

La definizione originaria dell'interfaccia **IFoo** era:

```
class IFoo {
     virtual void Func1(void) = 0;
     virtual void Func2(int nCount) = 0;
};
```

Questa però non è un interfaccia COM; per esserlo dobbiamo modificarla nel seguente modo:

```
interface IFoo : IUnknown {
     virtual HRESULT STDMETHODCALLTYPE Func1(void) = 0;
     virtual HRESULT STDMETHODCALLTYPE Func2(int nCount) = 0;
};
```

La parola *interface* non è una keyword in C++, infatti è definita (*#define*) come una struttura (*struct*). Questo perché in C++ le classi e le strutture sono la stessa cosa, eccetto per l'ereditarietà e per l'accesso di default ai membri che è sempre pubblico in entrambi i casi. La macro **STDMETHODCALLTYPE** indica al compilatore che deve generare la sequenza di chiamata standard della funzione.
Tutte le funzioni COM restituiscono un **HRESULT** (un numero a 32 bit) che serve per indicare se la funzione è andata a buon fine. Normalmente si ottiene il codice **S_OK** che indica il successo, ma se si verifica un errore, questo viene codificato e restituito nei 31 bit meno significativi.
Alla fine notiamo la derivazione da **IUnknown** che è la madre di tutte le interfacce COM. Questo significa che qualsiasi classe che implementa **IFoo** deve anche implementare l'interfaccia **IUnknown**. Quest'interfaccia contiene tre metodi: **QueryInterface**, **AddRef** e **Release** (che vedremo in seguito come implementare).
Riepilogando, un'interfaccia COM è una classe base astratta che eredita da **IUnknown**, le cui funzione hanno la sequenza di chiamata standard e restituiscono un **HRESULT**.
Un oggetto COM è di solito rappresentato come una scatola chiusa con dei pallini che indicano le interfacce che supporta (Figura 2.4).

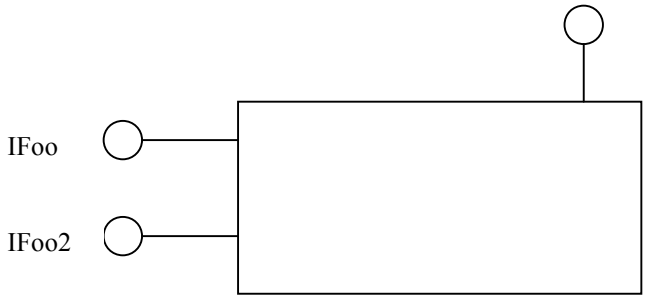


Figura 2.4 Un oggetto COM che implementa le interfacce **IFoo** e **IFoo2**. L'interfaccia **IUnknown** è senza etichetta.

2.6 Creare un oggetto

Una volta che il **CLSID** è associato ad un tipo di oggetto, possiamo crearne uno. È relativamente semplice, basta chiamare una funzione:

```
IFoo *pFoo = NULL;
HRESULT hr = CoCreateInstance(CLSID_Foo, NULL, CLSCTX_ALL,
                IID_IFoo, (void **) &pFoo);
```

Se la chiamata ha successo, crea un oggetto il cui tipo è specificato da **CLSID_Foo**. È da notare che non abbiamo un puntatore all'oggetto, ma un puntatore alla sua (una delle sue) interfaccia, per questo passiamo il parametro **IID_IFoo** che specifica il tipo di interfaccia che desidero. Il secondo parametro è importante per l'aggregazione che vedremo in seguito. Il terzo parametro indica il tipo del server (in questo caso il primo trovato). Si può usare ad esempio **CLSCTX_INPROC_SERVER** per una DLL, **CLSCTX_LOCAL_SERVER** per un EXE, e **CLSCTX_REMOTE_SERVER** per un server remoto. Nel nostro caso abbiamo usato **CLSCTX_ALL** che è un *or* dei casi precedenti, cioè COM cercherà prima una DLL, se non la trova cercherà un EXE. Se non trova nemmeno questo cercherà il server remoto il cui indirizzo è sempre specificato nel registro di Windows.
Una volta che si è ottenuto un puntatore all'interfaccia si possono chiamare i metodi nel seguente modo:

```
if (SUCCEDED(hr)) {
     pFoo->Func1();
     pFoo->Func2(5);
}
else // creazione fallita
```

CoCreateInstance restituisce un **HRESULT** per indicare il successo o il fallimento. Si può usare la macro **SUCCEDED** per controllare il risultato.

2.7 L'interfaccia IUnknown

L'interfaccia **IUnknown** definisce tre metodi che tutti gli oggetti COM devono implementare: **QueryInterface**, **AddRef** e **Release**. Quando sto creando un oggetto specifico anche l'interfaccia che intendo usare (lo abbiamo visto con **CoCreateInstance**), ma se il componente implementa diverse interfacce ci deve essere un modo per passare da un'interfaccia all'altra. La funzione **QueryInterface** svolge proprio questo compito.
Quando un client chiama un metodo di un componente, è COM che si occupa di caricarlo nello spazio logico del client (se è una DLL), o di mandare in esecuzione il corrispondente eseguibile (se è un EXE), ecc. Ma COM si occupa anche di scaricare la DLL e di terminare il processo quando il client (o i client) non hanno più bisogno del componente. Per fare ciò i client devono avere un metodo per avvisare COM

che non hanno più bisogno di un suo servizio. Questo metodo è il **reference counting** che è trattato tramite le funzioni **AddRef** e **Release**. La prima viene chiamata ogni volta che si ottiene un puntatore ad un'interfaccia, la seconda ogni volta che il puntatore non serve più. Vediamolo meglio con un esempio. Supponiamo di avere sempre il nostro solito componente che implementa le interfacce **IFoo** e **IFoo2** e che vogliamo utilizzarle tutte e due:

```
IFoo *pFoo = NULL;
HRESULT hr = CoCreateInstance(CLSID_Foo2, NULL, CLSCTX_ALL,
                IID_IFoo, (void **) &pFoo);
if (SUCCEDED(hr)) {
     pFoo->Func1(); // IFoo::Func1
     IFoo2 *pFoo2 = NULL;
     hr = pFoo->QueryInterface(IID_IFoo2, (void **) &pFoo2);
     if (SUCCEDED(hr)) {
          pFoo2->Func2Ex(5.5d); // IFoo2::Func2Ex
          pFoo2->Release();
     }
     pFoo->Release();
}
```

Notiamo che dobbiamo rilasciare tutti e due i puntatori quando non ci servono più. Non abbiamo mai usato **AddRef** perché è chiamata automaticamente all'interno di **QueryInterface** (anche **CoCreateInstance** chiama a sua volta **QueryInterface**). Se creiamo una copia di un puntatore ad interfaccia dobbiamo chiamare **AddRef** manualmente. Sbagliare il **reference counting** per difetto può provocare delle eccezioni nei client, per eccesso invece a degli sprechi di memoria. È importante chiamare **AddRef** e **Release** sullo stesso puntatore perché non sappiamo come è stato implementato il componente (potrebbe usare ad esempio un contatore per ogni interfaccia che supporta).
Siccome il **reference counting** è una cosa molto importante ma noiosa, ci sono molte classi di **smart pointer** che svolgono tutto il lavoro per noi (ne vedremo qualche esempio).

2.8 La Class Factory

La **Class Factory** o **Class Object** è un particolare oggetto COM che non è creato da **CoCreateInstance**, ma è sempre creato dalla funzione **CoGetClassObject**. COM memorizza in una tabella interna tutte le **Class Factory** e se viene richiesta una già esistente, invece di crearla, viene solo restituito un puntatore ad essa.
Lo scopo della **Class Object** è di fornire un metodo polimorfo standard per creare gli oggetti COM che vada bene sia per i server **InProc** che **OutOfProc**, senza che il client conosca i dettagli della creazione. Essa deve implementare l'interfaccia **IClassFactory** che contiene due funzioni: **CreateInstance** e **LockServer**.
Abbiamo visto come, quando si chiama **CoCreateInstance**, COM cerca nel registro il **CLSID** in modo da poter trovare la DLL o l'EXE che implementa il componente.

Vediamo adesso cosa succede in dettaglio. **CoCreateInstance** svolge i seguenti passi:

```
IClassFactory *pCF;
CoGetClassObject (rclsid, dwClsContext, NULL,
          IID_IClassFactory, (void **)&pCF);
HRESULT hr = pCF->CreateInstance(pUnkOuter, riid, ppvObj);
pCF->Release();
```

Il primo passo è ottenere una **Class Object** tramite un puntatore all'interfaccia **IID_IClassFactory**. Successivamente si chiama il metodo **CreateInstance** che si occupa di creare fisicamente il componente (i parametri usati sono gli stessi passati a **CoCreateInstance**). Alla fine si rilascia il puntatore al **Class Object** perché non lo si userà più. **CreateInstance** non ha tra i suoi parametri un **CLSID**; questo significa che abbiamo bisogno di almeno una **Class Object** per ogni diverso **CLSID** che vogliamo creare. La **Class Object** può implementare anche altre interfacce che, per esempio, oltre a preoccuparsi di creare l'oggetto, si occupano anche della sua inizializzazione.
Con questo schema di funzionamento, per il primo oggetto di un certo tipo che si vuole creare, COM ha molto lavoro da fare. Per prima cosa deve cercare nella sua tabella interna se la **Class Factory** è già stata creata. Se così non è, deve cercare nel registro usando come chiave il **CLSID** in modo tale da caricare la DLL o da avviare l'EXE. Alla fine COM chiama **IClassFactory::CreateInstance** sulla **Class Object** corretta per creare un'istanza dell'oggetto desiderato.
Se si deve creare una sola istanza di un oggetto, conviene chiamare **CoCreateInstance**. Ma se si devono creare molti oggetti dello stesso tipo, conviene ottenere (e conservare) un puntatore alla **Class Factory**. Per farlo è sufficiente chiamare **CoGetClassObject** e conservare il puntatore restituito per usarlo nella creazione degli oggetti successivi.
Se si conserva questo puntatore è necessario chiamare la funzione **IClassFactory::LockServer(TRUE)** per dire a COM di tenere in memoria il server. Non liberare il puntatore con **Release** non è sufficiente a tenere il server in memoria. Questa è una eccezione al normale comportamento di COM. Se non si blocca il server in memoria, si possono avere dei problemi di violazione di memoria (perché la DLL potrebbe essere stata scaricata, o l'EXE terminato). Quando il puntatore alla **Class Factory** non serve più si deve chiamare **Release** (sul puntatore) e la funzione **IClassFactory::LockServer(FALSE)**.

2.8.1 Implementare la Class Factory

Abbiamo visto che la **Class Factory** è un particolare oggetto COM che non viene creato da **CoCreateInstance** e che implementa almeno due interfacce: **IUnknown** e **IClassFactory**.
Una possibile dichiarazione della classe che la implementa potrebbe essere questa:

```
class CmyClassFactory : public IClassFactory
```

```
{
protected:
     ULONG m_cRef; // usato per il reference counting

public:
     CmyClassFactory() : m_cRef(0) { };

     // membri di IUnknown
     HRESULT QueryInterface(REFIID iid, void **ppv);
     ULONG AddRef(void);
     ULONG Release(void);

     // membri di IClassFactory
     HRESULT CreateInstance(IUnknown *pUnkOuter,
                            REFIID iid, void **ppv);
     HRESULT LockServer(BOOL fLock);
};
```

Come vediamo ci sono anche i metodi di **IUnknown** dato che **IClassFactory** deriva da questa (come tutte le interfacce COM del resto).
Ci sono molti modi per creare una **Class Object**, nessuno dei quali coinvolge **CoCreateInstance**. Nel nostro caso, poiché abbiamo veramente bisogno di una sola istanza di quest'oggetto e poiché si tratta di un oggetto di piccole dimensioni senza costruttore, possiamo crearlo come oggetto globale nel codice:

```
CmyClassFactory g_cfMyClassObject;
```

Questo significa che l'oggetto esisterà quando la DLL sarà caricata, o quando l'EXE sarà avviato. Abbiamo bisogno anche di un contatore globale che ci servirà per ricordare il numero di volte che è stata chiamata la funzione **IClassFactory::LockServer**, e per tenere traccia delle istanze degli oggetti che sono stati creati:

```
LONG g_cObjectsAndLocks = 0;
```

Implementiamo adesso i metodi della classe. Supponiamo che il nostro server sia realizzato come una DLL (nel caso di un EXE ci sono alcune differenze che prenderemo in esame nel momento in cui si presentano).

2.8.2 IUnknown::AddRef e IUnknown::Release

La nostra **Class Object** è globale. Essa esiste sempre e non può essere distrutta (almeno finché non viene scaricata la DLL o terminato l'EXE). Poiché non eliminiamo mai questo oggetto e poiché la referenza su questo oggetto non mantiene il server in memoria (come abbiamo visto in precedenza), non abbiamo bisogno di implementare il **reference counting**. Comunque, la specifica COM richiede che **Release** deve restituire zero quando non ci sono più puntatori attivi

all'oggetto (il nostro è un oggetto globale), quindi abbiamo bisogno di fare almeno una piccola implementazione.
AddRef e **Release** hanno la responsabilità di mantenere il **reference counting** sull'oggetto. Nella nostra classe abbiamo messo la variabile (un contatore) *m_cRef* che è inizializzata a zero, quindi **AddRef** e **Release** non devono fare altro che incrementare e decrementare questo contatore e restituire ogni volta il suo valore.
Se l'oggetto fosse stato creato dinamicamente, sarebbe stata **Release** ad occuparsi della cancellazione dell'oggetto (con *delete*), quando il contatore fosse uguale a zero ovviamente. Nel nostro codice non dobbiamo preoccuparci di ciò:

```
ULONG CmyClassFactory::AddRef() {
     return InterlockedIncrement(&m_cRef);
}

ULONG CmyClassFactory::Release() {
     return InterlockedDecrement(&m_cRef);
}
```

Siccome gli oggetti COM possono essere usati in **thread** diversi, ho usato le funzioni *Interlocked...* che sono thread-safe, invece di usare le semplici *++m_cRef* e *- -m_cRef*.

2.8.3 IUnknown::QueryInterface

L'implementazione di **QueryInterface** è standard perché si tratta di una **Class Object**. Quello che dobbiamo fare è controllare l'interfaccia richiesta (una tra **IUnknown** e **IClassFactory**) e, se si tratta di una di queste, restituiamo un puntatore all'interfaccia con un appropriato *cast*, altrimenti restituiamo un codice d'errore. Infine chiamiamo **AddRef** sul puntatore appropriato:

```
HRESULT CmyClassFactory::QueryInterface(REFIID iid,
                                        void **ppv) {
     *ppv = NULL;
     if (iid == IID_IUnknown || iid == IID_IClassFactory) {
          *ppv = static_cast<IClassFactory *> this;
          (static_cast<IClassFactory *> *ppv)->AddRef();
          return S_OK;
     }
     else {
          *ppv = NULL; // è richiesto dalla specifica
          return E_NOINTERFACE;
     }
}
```

Notiamo il nuovo operatore *static_cast*. Nell'ANSI C++ si possono avere tre differenti semantiche nell'uso del *cast*, usando diversi operatori. Con l'operatore

static_cast facciamo il *casting* appropriato su classi di tipo differente, cambiandone il valore se è necessario (non è questo il caso dato che abbiamo l'ereditarietà singola e non multipla).

2.8.4 IClassFactory::CreateInstance

Questo è il cuore della **Class Object**, la funzione che crea le istanze:

```
HRESULT CmyClassFactory::CreateInstance(IUnknown *pUnkOuter,
                          REFIID iid, void **ppv)
{
     *ppv = NULL;

     if (pUnkOuter != NULL)
          // non consideriamo l'aggregazione
          return CLASS_E_NOAGGREGATION;

     // creazione dell'oggetto
     CmyObject *pObj = new CmyObject();
     // è la classe che implementa il componente

     if (pObj == NULL) return E_OUTOFMEMORY;

     // ottiene il puntatore all'interfaccia richiesta
     HRESULT hr = pObj->QueryInterface(iid, ppv);

     // elimina l'oggetto se il puntatore non è disponibile
     if (FAILED(hr)) delete pObj;

     return hr;
}
```

Per prima cosa, siccome non consideriamo per adesso l'**aggregazione**, se il puntatore *pUnkOuter* è diverso da *NULL* restituiamo un errore perché ci è stato richiesto di aggregare il componente. Successivamente allochiamo l'oggetto, e restituiamo un errore se non riusciamo a farlo.
Con la chiamata a **QueryInterface** sul nuovo oggetto creato, otteniamo un puntatore all'interfaccia che ci è stata richiesta. Anche qui, se l'interfaccia non è supportata, restituiamo un errore. Come abbiamo visto in precedenza, è **QueryInterface** che chiama al suo interno **AddRef** in modo da avere un giusto **reference counting** sull'oggetto creato.
Non incrementiamo qui il contatore degli oggetti e dei lock *g_cObjectsAndLocks*. Dobbiamo farlo solo se la creazione ha successo e quindi il contatore lo gestiamo nel costruttore e distruttore dell'oggetto.

2.8.5 IClassFactory::LockServer

LockServer deve solo incrementare e decrementare il contatore globale di oggetti e lock *g_cObjectsAndLocks*. Non si deve preoccupare di scaricare la DLL quando il contatore è uguale a zero (se fossimo nel caso di un EXE, il server dovrebbe invece terminare):

```
HRESULT CmyClassFactory::LockServer(BOOL fLock)

{
     if (fLock)
          InterlockedIncrement(&g_cObjectsAndLocks);
     else
          InterlockedDecrement(&g_cObjectsAndLocks);
     return NOERROR;
}
```

Anche qui abbiamo reso il codice **thread-safe .**

2.9 Le altre funzioni della DLL

Vediamo come **CoGetClassObject** ottiene la **Class Factory**. Per i server **InProcess** questo è semplice: COM chiama una funzione chiamata **DllGetClassObject** che la DLL deve esportare. Il codice di **DllGetClassObject** è il seguente:

```
HRESULT DllGetClassObject(REFCLSID clsid, REFIID iid, void
**ppv)
{
     if (clsid != CLSID_MyObject) // è il giusto clsid?
          return E_FAIL;

     // ottiene l'interfaccia dall'oggetto globale
     HRESULT hr = g_cfMyClassObject.QueryInterface(iid, ppv);
     if (FAILED(hr)) *ppv = NULL;

     return hr;
}
```

COM passa a questa funzione un **CLSID** e un **IID**; **DllGetClassObject** restituisce un puntatore all'interfaccia richiesta in **ppv*. Se la **Class Object** non può essere creata o se l'interfaccia richiesta non esiste, si restituisce un errore. Dobbiamo controllare che il CLSID richiesto sia quello giusto, cioè che corrisponda a quello dell'oggetto che questa **Class Object** è in grado di creare.

Per i server EXE, il processo è differente. Infatti il server deve registrare (nella tabella interna di COM) una **Class Object** per ogni oggetto COM che può creare, chiamando la funzione **CoRegisterClassObject** per ognuna di esse. Quando il

processo termina deve chiamare la funzione **CoRevokeClassObject**, una per ogni **Class Object**, per rimuoverle dalla tabella degli oggetti registrati (lo vedremo in seguito).

Come COM ottiene la **Class Object** quando chiamiamo la funzione **CoGetClassObject**, dipende dal tipo di server. Se è una DLL, la carica (se non lo è ancora) e chiama **DllGetClassObject**. Se è un EXE, fa partire il processo (se non è già in esecuzione) e aspetta fino a che il server non ha registrato la **Class Object** che sta cercando, o fino a che non scade un **timeout**.

Un'altra funzione che dobbiamo far esportare alla nostra DLL è **DllCanUnloadNow**. COM chiamerà questa funzione per decidere se scaricare o meno la DLL. Nei server EXE il processo deve terminare quando non ci sono più client che interagiscono con noi, cioè quando la variabile *g_cObjectsAndLocks* è uguale a zero. Vediamo come è fatta la funzione **DllCanUnloadNow**:

```
HRESULT DllCanUnloadNow()
{
     if (g_cObjectsAndLocks == 0)
          return S_OK;
     else
          return S_FALSE;
}
```

Semplicemente restituiamo *S_OK* se non c'è più nessun oggetto attivo e nessun lock del server, *S_FALSE* altrimenti.
Abbiamo visto in precedenza che è necessario registrare il componente nel registro di Windows. Questo passo si può fare *a mano* utilizzando direttamente le utility che permettono di modificare il registro (**regedit.exe** per esempio), oppure si può scrivere del codice che si occupa di questo. Poiché una DLL conosce il componente (i componenti) che contiene, è lei stessa che può immettere le informazioni nel registro. Per fare ciò si devono esportare due funzioni:

DllRegisterServer
DllUnregisterServer

L'implementazione di queste due funzioni è semplice; infatti basta usare le funzioni di Win32 che permettono di modificare il registro (come ad esempio **RegOpenKeyEx**, **RegCreateKeyEx**, **RegSetValueEx**, **RegEnumKeyEx**, **RegDeleteKeyEx** e **RegCloseKeyEx**).
Si può usare il programma **regsvr32.exe** per richiamare automaticamente le funzioni che registrano e deregistrano i componenti.

2.10 Modifiche per i server OutOfProc

Gli EXE non possono esportare funzioni. Al momento abbiamo visto che i nostri server **InProc** dipendono dalle seguenti funzioni esportate:

DllCanUnloadNow
DllGetClassObject
DllRegisterServer
DllUnregisterServer

Ci occorrono dei rimpiazzi per queste funzioni. Sostituire la prima è facile. A differenza di una DLL, un EXE non è passivo: esso controlla la propria esistenza. L'EXE può monitorare il contatore dei lock e scaricarsi dalla memoria quando questo raggiunge lo zero. Pertanto, gli EXE non devono implementare questa funzione.
Le ultime due sono altrettanto facilmente sostituibili. Gli EXE possono supportare l'autoregistrazione accettando dei parametri da riga di comando (**RegServer** e **UnRegServer**). Tutto ciò che il server deve fare è gestire queste due opzioni. Possiamo per esempio scrivere due funzioni **RegisterAll** e **UnregisterAll** che si occuperanno di queste opzioni (e saranno implementate come le corrispondenti della DLL, tranne che per la chiave che indica il tipo di server che sarà **LocalServer32** invece di **InprocServer32**).
La funzione **DllGetClassObject** è più difficile da sostituire. In precedenza abbiamo visto che **CoCreateInstance** richiama **CoGetClassObject** che a sua volta richiama **DllGetClassObject**. Questa restituisce un puntatore a **IClassFactory** che viene usato per creare il componente. Poiché l'EXE non può esportare **DllGetClassObject**, occorre un altro metodo per ottenere il puntatore a **IClassFactory**. La soluzione COM è quella di conservare una tabella privata delle **Class Factory** registrate.
Quando il client richiama **CoGetClassObject** con i parametri corretti, COM dapprima verifica in questa tabella interna di **Class Factory** se esiste il **CLSID** richiesto. Se non la trova, COM consulta il registro e avvia l'EXE associato. È compito di quest'ultimo di registrare (nella tabella interna) le **Class Factory** non appena possibile, in modo che COM possa trovarle. La registrazione avviene semplicemente creando la **Class Factory** e passando il suo puntatore a interfaccia alla funzione **CoRegisterClassObject**. Questo si deve fare per ognuna delle **Class Factory** che il server supporta.
Supponiamo di avere un array *g_FactoryDataArray* che contiene le informazioni delle **Class Factory** che vogliamo registrare. Appena partito l'EXE dovrebbe chiamare una funzione del genere:

```
BOOL CFactory::StartFactories()
{
   CFactoryData* pStart = &g_FactoryDataArray[0];
   CFactoryData* pEnd =
      &g_FactoryDataArray[g_cFactoryDataEntries - 1];

   for(CfactoryData* pData = pStart; pData <= pEnd; pData++)
   // inizializza il puntatore alla Class Factory
   // e al cookie
```

```
      pData->m_pIClassFactory = NULL;
      pData->m_dwRegister = NULL;

      // crea la Class Factory per questo componente
      IClassFactory* pIFactory = new CFactory(pData);

      // registra la Class Factory
      DWORD dwRegister;
      HRESULT hr = ::CoRegisterClassObject(*pData->m_pCLSID,
         static_cast<IUnknown*>(pIFactory),
         CLSCTX_LOCAL_SERVER,
         REGCLS_MULTIPLEUSE,
         &dwRegister);
      if (FAILED(hr))
      {
         pIFactory->Release();
         return FALSE;
      }

      // imposta i dati
      pData->m_pIClassFactory = pIFactory;
      pData->m_dwRegister = dwRegister;
   }
   return TRUE;
}
```

La variabile *m_pIClassFactory* conserva il puntatore alla **Class Factory** in esecuzione per il **CLSID** memorizzato in *m_pCLSID*. La variabile *m_dwRegister* conserva il **cookie** che servirà per eliminare la **Factory** da quelle registrate.
I parametri di **CoRegisterClassObject** sono quindi: il **CLSID** della classe che vogliamo creare, il puntatore alla sua **Factory**, due **flag** e una variabile che conterrà le informazioni che restituisce la funzione (**cookie**).
Vediamo il significato dei due **flag**. Questi vengono utilizzati insieme, e il significato di uno cambia in base all'altro. Il quarto parametro ci dice se un'unica copia in esecuzione dell'EXE può servire più di un esemplare attivo di un componente. Se il server può servire solo un componente, dobbiamo utilizzare **REGCLS_SINGLEUSE** e **CLSCTX_LOCAL_SERVER**. Se invece il server può supportare diversi esemplari dei suoi componenti, dobbiamo utilizzare **REGCLS_MULTI_SEPARATE**. Questo produce una situazione interessante. Supponiamo di avere un EXE che registra alcuni componenti, e supponiamo che debba utilizzare uno dei componenti che sta registrando (nota bene: non è il client che usa il componente, ma è lo stesso server che lo crea, ad usarlo). Se registrassimo la **Factory** con l'opzione precedente, verrebbe caricata un'altra copia dell'EXE per servire le richieste del server originale. Questo ovviamente non è quasi mai il metodo più efficiente che possiamo desiderare.
Per registrare il server EXE come server **InProc** per le sue chiamate ai suoi componenti e come server **OutOfProc** per le chiamate dei client ai suoi componenti, si deve fare una chiamata del genere:

```
hr = ::CoRegisterClassObject(clsid, pIUnknown,
```

```
      (CLSCTX_LOCAL_SERVER | CLSCTX_INPROC_SERVER),
      REGCLS_MULTISEPARATE, &dwRegister);
```

Combinando i due **flag**, il server EXE può servire se stesso per i propri componenti. Dato che questo è il caso più frequente, si utilizza un **flag** particolare, **REGCLS_MULTIPLEUSE**, che abilita automaticamente **CLSCTX_INPROC_SERVER** anche quando non è settato. Quindi la seguente chiamata e equivalente alla precedente:

```
hr = ::CoRegisterClassObject(clsid, pIUnknown,
          CLSCTX_LOCAL_SERVER,
          REGCLS_MULTIPLEUSE,
          &dwRegister);
```

Quando il server termina deve rimuovere le **Class Factory** che ha registrato dalla tabella interna di COM. La funzione della libreria COM che svolge questo compito è **CoRevokeClassObject**. Seguendo l'esempio precedente le **Factory** si possono rimuovere con una funzione del tipo:

```
void CFactory::StopFactories()
{
   CFactoryData* pStart = &g_FactoryDataArray[0];
   CFactoryData* pEnd =
      &g_FactoryDataArray[g_cFactoryDataEntries - 1];

   for(CFactoryData* pData = pStart; pData <= pEnd; pData++)
   {
      // ottiene il cookie e ferma l'esecuzione della Factory
      DWORD dwRegister = pData->m_dwRegister;
      if (dwRegister != 0)
         ::CoRevokeClassObject(dwRegister);

      // rilascia la Class Factory
      IClassFactory* pIFactory = pData->m_pIClassFactory;
      if (pIFactory != NULL)
         pIFactory->Release();
   }
}
```

I server **InProc** esportano la funzione **DllCanUnloadNow**; questa viene chiamata da COM per sapere se può scaricare la DLL dalla memoria. Siccome un EXE ha il controllo di se, può decidere da solo quando terminare. Quindi si deve modificare la funzione **LockServer** nel seguente modo:

```
HRESULT CMyClassObject::LockServer(BOOL fLock)

{
   if (fLock)
      InterlockedIncrement(&g_cObjectsAndLocks);
```

```
   else
      InterlockedDecrement(&g_cObjectsAndLocks);
   // termina il server se non serve più
   if (g_cObjectsAndLocks == 0)
      CloseExe();
   return NOERROR;
}
```

Ora dovrebbe essere chiaro del perché non contano le **Class Factory** tra i componenti attivi. La prima cosa che un server locale fa, è quella di creare tutte le proprie **Class Factory**, l'ultima cosa è quella di chiuderle tutte. Quindi il server non deve attendere che le **Class Factory** siano rilasciate prima di scaricarsi, perché è lui stesso che deve farlo. Pertanto, i client utilizzano la funzione **IClassFactory::LockServer** se vogliono garantirsi che il server rimanga in memoria mentre stanno tentando di creare componenti.

2.11 Implementare il componente

Abbiamo visto che all'interno di **CoCreateInstance** il componente vero e proprio era creato con l'istruzione:

```
CMyObject *pObj = new CMyObject;
```

Ora vedremo il codice della classe *CMyObject* che rappresenta il componente.

Il nostro oggetto implementerà quattro interfacce: **IFoo**, **IFoo2**, **IGoo** e, naturalmente, **IUnknown**. **IFoo2** è un'estensione di **IFoo**, **IFoo** più una nuova funzione (che dovrebbe rappresentare una nuova versione dell'interfaccia, un *upgrade*), mentre **IGoo** è un'interfaccia completamente separata. Così il componente può essere schematizzato con il diagramma della Figura 2.5.

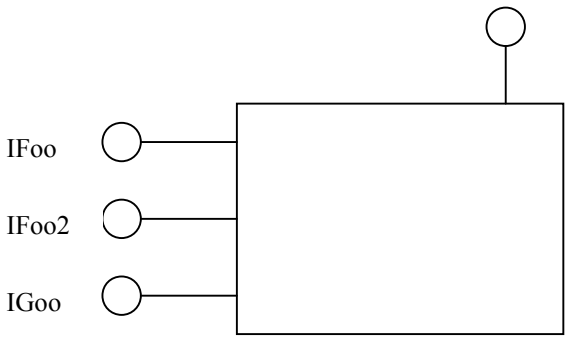


Figura 2.5 Diagramma del componente con le sue interfacce.

È da notare che, come implica il diagramma, COM e più specificamente **QueryInterface**, non considera **IFoo** e **IFoo2** in relazione, come del resto non considera in relazione **IUnknown** con le altre interfacce. Quando si richiede

IUnknown, si ottiene proprio **IUnknown**, e quando si richiede un'altra interfaccia, si ottiene proprio quella.

Vediamo adesso la dichiarazione delle tre interfacce:

```
interface IFoo : IUnknown {
     virtual HRESULT STDMETHODCALLTYPE Func1(void) = 0;
     virtual HRESULT STDMETHODCALLTYPE Func2(int nCount) = 0;
};
```

Ma per semplificare le cose, riscriviamola con delle macro che sono definite negli *header* della libreria COM:

```
interface IFoo : IUnknown {
     STDMETHOD Func1(void) PURE;
     STDMETHOD Func2(int nCount) PURE;
};
```

Le altre due sono:

```
interface IFoo2 : IFoo {
     STDMETHOD Func3(int *pout) PURE;
};

interface IGoo : IUnknown {
     STDMETHOD Gunc(void) PURE;
};
```

L'interfaccia **IFoo2** ha sei metodi: i tre di **IUnknown**, i due di **IFoo** e *Func3*. La stessa cosa vale per **IFoo** e **IGoo** che hanno rispettivamente cinque e quattro metodi.

Le interfacce devono essere documentate all'esterno del sorgente; e quindi spieghiamo cosa fanno le nostre interfacce.

Quando il componente è creato, avrà un valore interno che vale 5. La funzione *Func1* incrementerà questo valore interno e farà fare un *beep* allo speaker del PC se il valore è un multiplo di 3. La funzione *Func2* setterà il valore interno col parametro che gli viene passato. Non abbiamo inserito nessun metodo per leggere il valore interno dell'oggetto, ma supponiamo che, dopo aver venduto il nostro componente, la mancanza di questo metodo si faccia sentire. Non dobbiamo fare altro che definire una nuova interfaccia (derivata da **IFoo**) che ne preveda uno (proprio quello che abbiamo fatto con l'interfaccia **IFoo2** definendo il metodo *Func3*). Così i vecchi client continueranno a funzionare usando la vecchia interfaccia, mentre i nuovi potranno avvalersi delle potenzialità della nuova interfaccia. Vogliamo anche che il componente abbia la capacità di far fare *beep* allo speaker ogni volta che vogliamo, così abbiamo aggiunto l'interfaccia **IGoo** con il suo metodo *Gunc* che fa proprio questo.

Per come abbiamo ideato il nostro componente, l'interfaccia **IGoo** non dipende dallo stato interno del componente. Sono le altre due interfacce che interagiscono con esso. Questo significa che qualsiasi oggetto può implementare **IGoo** senza implementare le altre due interfacce, e viceversa. In altre parole queste interfacce non sono unite.

2.11.1 Come implementare l'oggetto

Il nostro componente deve implementare quattro interfacce: **IFoo**, **IFoo2**, **IGoo** e **IUnknown**. Ci sono vari modi per implementare il componente. Uno potrebbe essere per esempio implementare ogni interfaccia con un oggetto separato. Se così fosse, la funzione **QueryInterface** dovrebbe allora restituire un puntatore all'oggetto appropriato. Una variazione potrebbe essere inserire questi oggetti come membri di un altro oggetto e crearli come classi annidate. La **MFC** usa questo metodo internamente. Comunque il metodo più *pulito* è fare ereditare al componente da tutte le interfacce che vogliamo implementare (come fa la **ATL**, Active Template Library, una serie di classi e template sviluppate da Microsoft per semplificare lo sviluppo di oggetti COM in C++). Noi vedremo quest'ultimo metodo. Tutto quello che deve fare **QueryInterface**, in questo caso, è restituire un puntatore alla appropriata classe base (dopo aver fatto il relativo *casting*).

Così la dichiarazione della nostra classe di implementazione sarà:

```
class CmyObject :
     public IFoo2,
     public IGoo
{
private:
     int m_iInternalValue;
     ULONG m_refCnt;

public:
     CMyObject();
     virtual ~CMyObject();

     // IUnknown
     STDMETHODIMP QueryInterface(REFIID, void **);
     STDMETHODIMP_(ULONG) AddRef(void);
     STDMETHODIMP_(ULONG) Release(void);

     // IFoo
     STDMETHODIMP Func1(void);
     STDMETHODIMP Func2(int nCount);

     // IFoo2
     STDMETHODIMP Func3(int *pout);

     // IGoo
```

```
    STDMETHODIMP Gunc(void);
};
```

È da notare che non ereditiamo esplicitamente da **IUnknown** o **IFoo**. Non ereditiamo da **IFoo** perché **IFoo2** eredita da essa. Per la stessa ragione non ereditiamo da **IUnknown** perché sia **IFoo2** che **IGoo** ereditano da essa (direttamente o indirettamente).

Se avessimo ereditato da **IFoo** o **IUnknown** direttamente, avremmo creato un'ambiguità e avremmo ottenuto un errore da parte del compilatore. Quindi la regola è la seguente: si deve ereditare solo dalla classe *più derivata*. Comunque si devono implementare tutti i metodi, anche quelli delle classi base (Figura 2.6).

Come possiamo vedere ci sono due cammini che portano a **IUnknown**. In teoria ci dovrebbero essere dei problemi di ambiguità, ma non è così. Vediamo il perché. Se avessimo ereditato l'implementazione di **IUnknown**, avremmo dei problemi perché ci troveremmo con due implementazioni dello stesso oggetto. Ma nel nostro caso non è così. La classe **IUnknown** dalla quale ereditiamo è una classe virtuale pura, così ereditiamo solo l'interfaccia.

Se si eredita solo l'interfaccia da una classe, normalmente si scriverà solo un'implementazione per queste funzioni, tipicamente per quelle della classe più derivata. Così dobbiamo scrivere solo un'implementazione di **QueryInterface**, **AddRef** e **Release**.

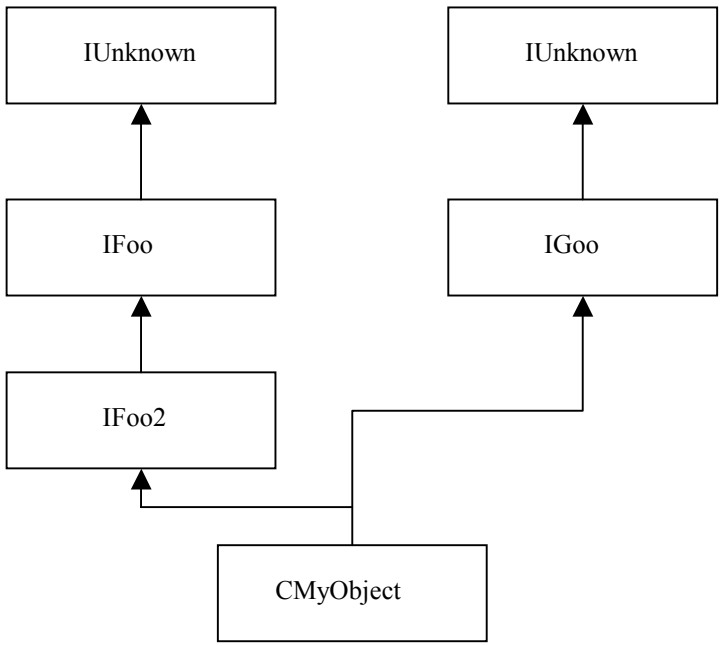


Figura 2.6 Diagramma dell'ereditarieta dell'oggetto.

Tutti i puntatori delle *vtable* di **IUnknown** punteranno alle stesse funzioni **QueryInterface**, **AddRef** e **Release**. Questo è completamente consistente con il comportamento delle funzioni virtuali del C++: quando si chiama una funzione

virtuale, si chiama sempre l'implementazione più derivata disponibile. Vediamo una possibile implementazione della *vtable* del nostro oggetto nella Figura 2.7 (una possibile implementazione perché la *vtable* dipende dal compilatore che si sta usando).

Tutti i puntatori della *vtable*, anche quelli della sezione di **IUnknown** di **IGoo**, puntano alle funzioni di *CMyObject*, così si è sicuri di chiamare le implementazioni più derivate.

Se da una parte questo comportamento è una vera panacea per l'implementazione di **IUnknown**, dall'altra può creare dei problemi se per caso differenti interfacce hanno dei metodi con i nomi uguali. In questo caso, implementando l'oggetto COM con l'ereditarietà multipla, sorgerebbero dei problemi per distinguere i metodi che hanno il nome uguale, cioè il compilatore accetterebbe solo un'implementazione che verrebbe poi chiamata per ogni interfaccia che esporta questa funzione. Questo è il principale svantaggio nell'implementare gli oggetti COM con l'ereditarietà multipla. Per fortuna questo accade raramente e, se accadesse, il problema si potrebbe risolvere implementando questo oggetto con le classi annidate (come abbiamo visto prima) e gli altri (in cui non ci sono conflitti di nomi) con l'ereditarietà multipla.
Facendo il *casting* del puntatore *this* a **IUnknown***, **IFoo***, **IFoo2*** o **CMyObject**, non ne cambierà il valore, solo il *casting* a **IGoo** cambierà il valore del puntatore.

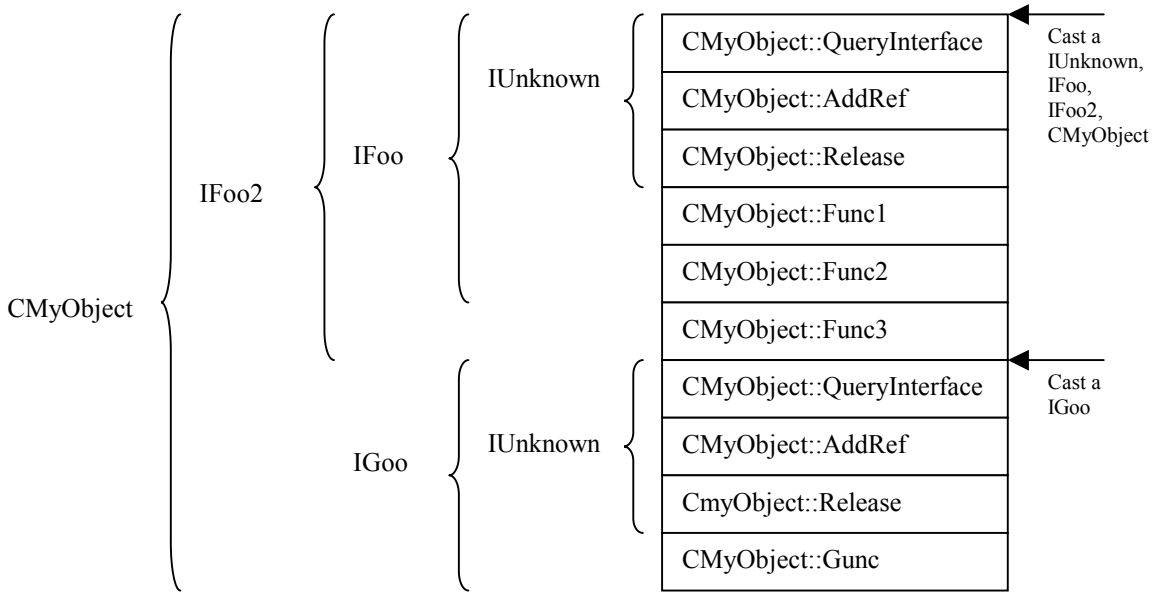


Figura 2.7 Una possibile implementazione della *vtable* di **CMyObject.**

Quindi restituiremo il puntatore *this* con il *cast* a **IFoo2*** quando verrà chiesto a **QueryInterface** il puntatore alle interfacce **IUnknown**, **IFoo** e **IFoo2**, mentre restituiremo *this* con il *cast* a **IGoo*** se verrà chiesto il puntatore a **IGoo** (questo comporterà la modifica del puntatore che punterà così alla seconda *vtable* (vedi Figura 2.8).

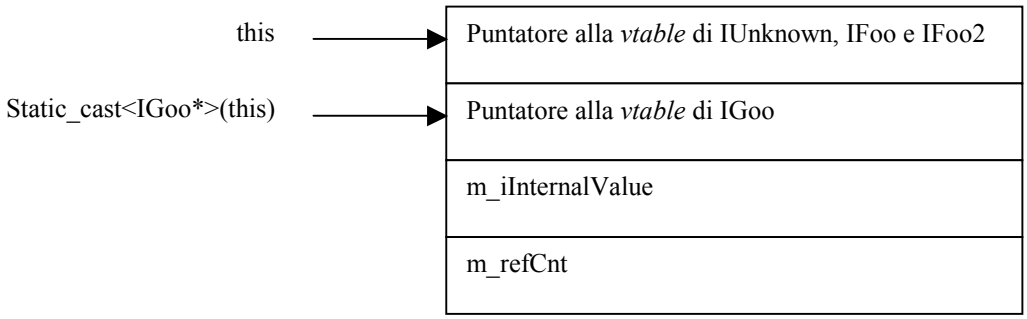


Figura 2.8 Rappresentazione in memoria di **CMyObject.**

2.11.2 CMyObject::QueryInterface

Adesso che sappiamo cosa deve fare **QueryInterface**, diamo un occhiata al codice:

```
STDMETHODIMP CMyObject::QueryInterface(REFIID iid,
void **ppv)
{
     if (ppv == NULL) return E_INVALIDARG;
*ppv = NULL;
     if (iid == IID_IUnknown ||
          iid == IID_IFoo ||
          iid == IID_IFoo2)
     {
          *ppv = static_cast<IFoo2*>(this);
     }
     else if (iid == IID_IGoo)
     {
          *ppv = static_cast<IGoo*>(this);
     }
     if (*ppv)
     {
          AddRef();
          return S_OK;
     }
     else return E_NOINTERFACE;
}
```

Chiamiamo **AddRef** se **QueryInterface** ha successo. Ogni volta che si restituisce un puntatore a interfaccia si deve chiamare **AddRef** su di esso (e il client deve chiamare **Release** quando ha finito di usarlo).

Quando il client richiede **IID_IGoo**, chiamiamo **AddRef** con un differente puntatore da quello restituito al client. Per supportare il **reference counting** *per interfaccia*, i client COM devono chiamare **AddRef** e **Release** sullo stesso puntatore. Nel nostro caso (siccome abbiamo una certa conoscenza di come è implementato il **reference counting** nel nostro oggetto) possiamo anche non essere così pignoli e infatti chiamiamo *this->AddRef()* che funziona benissimo. In una

situazione più complicata, come per esempio nelle interfacce implementate con le classi annidate, non possiamo fare questa semplificazione (e quindi se ci è stato chiesto un puntatore ad **IGoo**, dovremmo chiamare **AddRef** sullo stesso puntatore che restituiamo al client).

2.11.3 CMyObject::AddRef e CMyObject::Release

Le implementazioni di **AddRef** e **Release** sono semplici:

```
STDMETHODIMP_(ULONG) CMyObject::AddRef(void)
{
     return ++m_refCnt; // non è thread-safe
}

STDMETHODIMP_(ULONG) CMyObject::Release(void)
{
     --m_refCnt; // non è thread-safe
     if (m_refCnt == 0)
     {
          delete this;
          return 0;
     }
     else return m_refCnt;
}
```

Se usiamo i componenti in un client multi-threaded, dobbiamo usare invece degli operatori ++ e --, le due funzioni **InterlockedIncrement** e **InterlockedDecrement** che sono **thread-safe**.

Oltre a decrementare il contatore, **Release** ha la responsabilità di distruggere l'oggetto quando il contatore arriva a zero. Se questo accade, deve restituire zero per indicare che l'oggetto non esiste più.

2.11.4 Il costruttore e il distruttore

Questo è il codice del costruttore e del distruttore:

```
CMyObject::CMyObject() : m_iInternalValue(5), m_refCnt(1)
{
     g_cObjectsAndLocks++; // non è thread-safe
}

CMyObject::~CMyObject()
{
     g_cObjectsAndLocks--; // non è thread-safe
}
```

Il costruttore e il distruttore nascondono due piccoli trucchi. Il primo è che il costruttore inizializza il **reference count** a uno e non a zero. La spiegazione di ciò si deve cercare nel codice di **CMyClassObject::QueryInterface**, la quale chiama **Release** sull'oggetto dopo che ha fatto la prima **QueryInterface**. Se la prima **QueryInterface** ha successo, chiama **AddRef**, che incrementa il **reference count**, lasciandolo a due. Se fallisce, **AddRef** non è chiamata, così il **reference count** rimane a uno. Ma in entrambi i casi, **CreateInstance** chiama **Release** che decrementa il contatore. Se la prima **QueryInterface** ha successo, il contatore vale quindi uno (uno del costruttore più una **AddRef** , meno una **Release**); se fallisce il contatore vale zero e **Release** distrugge l'oggetto.
Il secondo trucco è che il costruttore e il distruttore incrementano e decrementano il contatore globale di oggetti e lock. Questo contatore è inizializzato a zero quando la DLL è caricata, ed è incrementato per ogni oggetto creato (esclusa la **Class Factory**) e per ogni chiamata a **IClassFactory::LockServer(TRUE)**. È decrementato quando ogni oggetto è distrutto e per ogni chiamata a **IClassFactory::LockServer(FALSE)**.

Ovviamente è possibile spostare l'incremento in **CreateInstance**, e il decremento in **Release** se si distrugge l'oggetto, ma la soluzione precedente è più elegante.

2.11.5 Implementare le interfacce custom

Tutto quello che abbiamo visto fino ad adesso è l'*overhead* che è necessario per sviluppare oggetti COM (vedremo come è possibile semplificare tutto questo). Adesso concentriamoci sulle interfacce che vogliamo che il nostro componente deve supportare:

```
// IFoo
STDMETHODIMP CMyObject::Func1()
{
     m_iInternalValue++;
     if (m_iInternalValue % 3 == 0)
          MessageBeep((UINT) -1);
     return S_OK;
}

STDMETHODIMP CMyObject::Func2(int nCount)
{
     m_iInternalValue = nCount;
     return S_OK;
}

// IFoo2
STDMETHODIMP CMyObject::Func3(int *pout)
{
     *pout = m_iInternalValue;
     return S_OK;
```

```
}

// IGoo
STDMETHODIMP CMyObject::Gunc(void)
{
     MessageBeep((UINT) -1);
     return S_OK;
}
```

2.12 IDL e MIDL

Uno dei requisiti fondamentali dei componenti è l'indipendenza dal linguaggio, ma negli esempi precedenti abbiamo scritto tutto in C++. Come è possibile allora far interagire client e server se sono scritti in linguaggi diversi?
L'unica cosa in comune che hanno il client ed il server è l'interfaccia; occorre quindi un metodo per definire le interfacce, indipendentemente dal linguaggio usato per implementarle. Il metodo usato da COM è il linguaggio **IDL** (**Interface Definition Language**). Il linguaggio **IDL**, come la struttura degli **UUID** e la specifica **RPC**, derivano dal **DCE** (**Distributed Computing Environment**) dell'**Open Software Foundation**.
Con una sintassi simile a quella di C e C++, l'**IDL** viene usato per descrivere in modo completo le interfacce e i dati condivisi dal client e dal componente. Comunque le interfacce COM utilizzano soltanto un sottoinsieme dell'**IDL**, ed hanno bisogno di diverse estensioni non standard che la Microsoft ha aggiunto per supportare COM. È nato così il **Microsoft IDL**, comunemente chiamato **MIDL**. Vediamo la traduzione delle nostre interfacce in **MIDL**:

```
[
     object,
     uuid(4ECECC21-D25E-11d2-8B40-00400559C94F),
     helpstring("IFoo interface"),
     pointer_default(unique)
]
interface IFoo : IUnknown
{
     HRESULT Func1();
     HRESULT Func2([in] int nCount);
};

[
     object,
     uuid(4ECECC22-D25E-11d2-8B40-00400559C94F),
     helpstring("IFoo2 interface"),
     pointer_default(unique)
]
interface IFoo2 : IFoo
{
```

```
      HRESULT Func3([out, retval] int *pout);
};

[
      object,
      uuid(4ECECC23-D25E-11d2-8B40-00400559C94F),
      helpstring("IGoo interface"),
      pointer_default(unique)
]
interface IGoo : IUnknown
{
      HRESULT Gunc();
};
```

Possiamo notare come la sintassi MIDL non sia poi tanto diversa da quella C++. La differenza più ovvia riguarda le informazioni delimitate dalle parentesi quadre. Ogni interfaccia ha una lista di attributi o una intestazione di interfaccia che precede il corpo dell'interfaccia stessa.
La prima parola chiave *object*, significa che si deve compilare l'interfaccia come un oggetto COM, e non come una interfaccia **RPC**. La seconda parola chiave *uuid* specifica l'**IID** di questa interfaccia. La terza parola chiave *helpstring* viene usata per inserire una stringa di help nella **type library**, che è una descrizione in formato binario delle interfacce (vedremo a cosa servono). La quarta parola chiave *pointer_default* è un po' meno chiara, e quindi ha bisogno di qualche spiegazione.
Uno degli scopi di **IDL** è quello di fornire informazioni tali che si possa fare il **marshaling** dei parametri delle funzioni. Per questo motivo, **IDL** deve avere informazioni su come trattare cose come i puntatori. La parola chiave *pointer_default* indica il modo di trattare i puntatori, che possono apparire nei parametri delle funzioni, quando non è stato dato nessun altro tipo di attributo per un puntatore. Essa dispone di tre diverse opzioni:

- **ref**: i puntatori vengono trattati come riferimenti. Essi punteranno sempre ad un indirizzo valido e potranno sempre essere dereferenziati. Non potranno essere *NULL*. Punteranno alla stessa locazione di memoria prima e dopo una chiamata. Inoltre non potranno avere degli alias all'interno della funzione.
- **unique**: questi puntatori possono essere *NULL*. Possono anche cambiare all'interno di una funzione. Tuttavia non possono avere alias all'interno della funzione.
- **ptr**: questa opzione specifica che il puntatore di default è equivalente al puntatore C. Il puntatore può anche avere un alias, può essere *NULL* e può cambiare.

Queste informazioni servono per ottimizzare il **marshaling**.

Di tutti gli attributi delle interfacce e di altri che possono essere presenti (come ad esempio *local*, *id*, *dual*, ecc.) solo l'*uuid* è strettamente necessario, gli altri sono opzionali.

Vediamo anche il significato degli attributi *[in]*, *[out]*, *[in, out]* e *[retval]* che precedono i tipi dei parametri. **MIDL** utilizza anche gli attributi di parametro *in* e

out per ottimizzare ulteriormente la traduzione dei parametri. Se un parametro è marcato come *in*, **MIDL** sa che quel parametro deve solo essere trasferito dal client al server. Il codice **stub** non ha bisogno di restituire nessuna informazione. La parola chiave *out* dice a **MIDL** che il parametro viene usato solo per restituire dati dal server al client. Il **proxy** non ha bisogno di effettuare il **marshaling** su un parametro *out* e di inviarlo al server. I parametri possono anche essere marcati con entrambe le parole chiave. L'attributo *retval* indica che quel parametro è un valore di ritorno.

Un discorso particolare si deve fare se i parametri sono stringhe. Per fare il **marshaling** di un particolare dato, occorre sapere quanto spazio occupa, per poterlo copiare. È facile stabilire quanto è lunga una stringa C++, basta cercare il carattere *NULL* finale. Collocando il modificatore *string* ad un parametro (supponiamo ad esempio *[in, string]* oppure *[out, string]*, **MIDL** sa che il parametro è una stringa, e che può determinarne la lunghezza cercando appunto il carattere *NULL* finale. La convenzione standard di COM per le stringhe è quella di utilizzare i caratteri Unicode *wchar_t*, anche in sistemi operativi come Microsoft Windows 95/98, che non gestiscono in modo nativo Unicode.

Una volta definite le interfacce in **MIDL** le passiamo ad un compilatore, il programma **MIDL.EXE**, che traduce le interfacce definite in **MIDL** nelle corrispondenti C/C++, ed inoltre genera anche altri file che servono per semplificare il lavoro del programmatore COM. Supponiamo di avere un file *foo.idl* che contiene la definizione di alcune interfacce. Il file verrà compilato con la seguente riga di comando:

```
midl foo.idl
```

Questo comando genererà i seguenti file:

- **foo.h**: un file di intestazione (compatibile con C e C++) che contiene le dichiarazioni di tutte le interfacce descritte nel file *foo.idl*. Include anche tutti i file di intestazione della libreria di COM che sono necessari (definizioni di macro e codici di errore). Il nome di questo file può essere modificato utilizzando uno degli switch */header* o */h* tra loro equivalenti.
- **foo_i.c**: un file C che contiene tutti i **GUID** utilizzati nel file *foo.idl*. Inoltre associa a questi **GUID** anche dei corrispondenti nomi simbolici (ad esempio **IID_IFoo**). Il nome di questo file può essere modificato usando lo switch */iid*.
- **foo_p.c**: un file C che implementa il codice **proxy** e **stub** per le interfacce del file *foo.idl*. Il nome di questo file può essere modificato usando lo switch */proxy*.
- **dlldata.c**: un file C che implementa la DLL che contiene il codice **proxy** e **stub**. Il nome di questo file può essere modificato usando lo switch */dlldata*.
- **foo.tlb**: è la **type library** associata al file *foo.idl*. Questo file viene generato solamente se nel file *.idl* è presente un blocco marcato con la parola chiave *library*. Contiene la definizione delle interfacce in formato binario. È l'equivalente di un file di intestazione C/C++, indipendente dal linguaggio, che può essere usato dai linguaggi interpretati e dagli ambienti di programmazione macro. Una **type library** è una versione compilata di un file **MIDL**, alla quale si

può accedere da programma. Gli ambienti di sviluppo tipo **Visual Basic**, **Visual J++**, utilizzano questo file per utilizzare le interfacce (e non il file foo.h, dato che è in linguaggio C).

La Figura 2.9 mostra i file generati dal compilatore **MIDL** più il file *foo_impl.cpp* che contiene il codice per l'implementazione delle interfacce e delle altre funzioni che la DLL deve esportare. Mostra inoltre anche i passi necessari per poter ottenere un server **InProc** (la DLL) che implementa il componente.

Riepilogando, per ottenere il componente **InProc** che abbiamo descritto in precedenza, mettiamo la definizione delle interfacce in **MIDL** nel file *componente.idl*. Includiamo però all'inizio due file di intestazione standard di COM con la direttiva *import*:

```
import "oaidl.idl";
import "ocidl.idl";
```

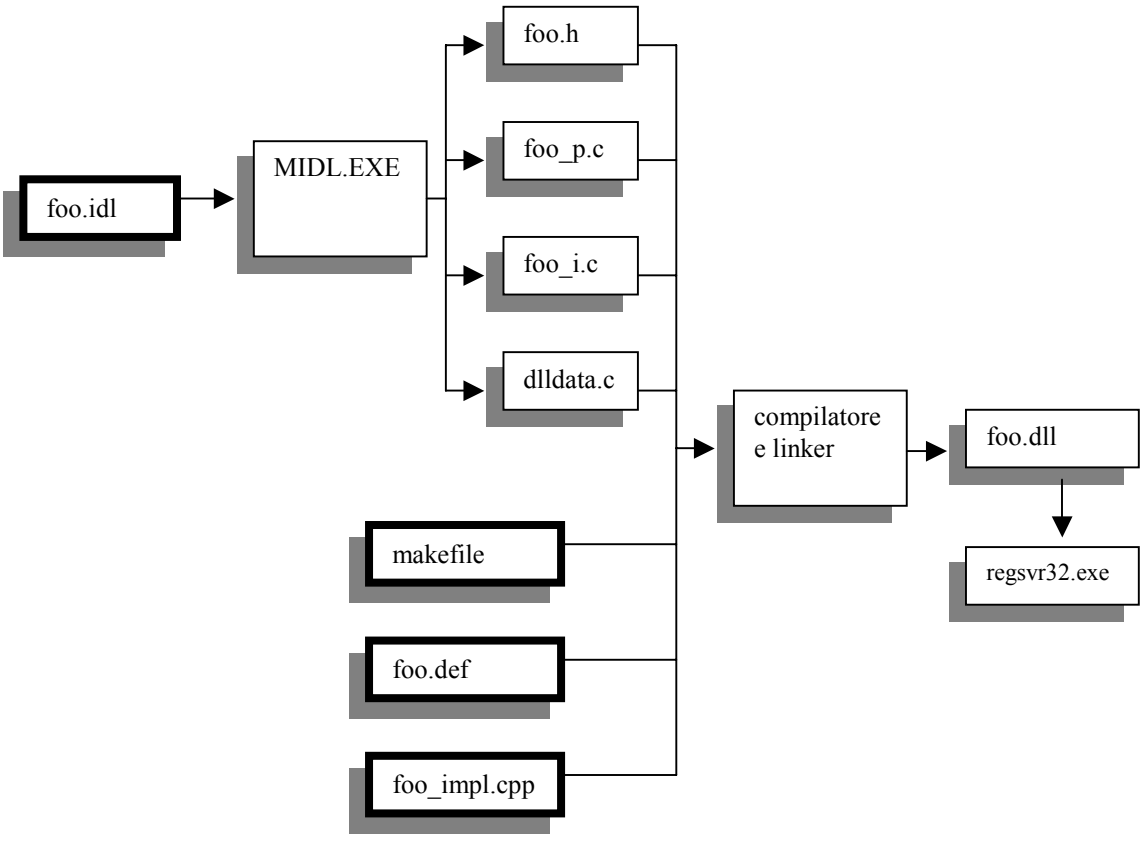


Figura 2.9 Processo che mostra i passi necessari per ottenere un componente **InProc**. I file forniti dal programmatore sono marcati in nero.

Alla fine del file aggiungiamo un blocco con la direttiva *library* per creare anche una **type library**:

```
[
     uuid(2A809781-D29B-11d2-8B40-00400559C94F),
     version(1.0),
     helpstring("Componente 1.0 Type Library")
]
library COMPONENTELib
{
     importlib('stdole32.tlb');
     importlib('stdole2.tlb');

     [
          uuid(BF32B161-D29B-11d2-8B40-00400559C94F)
     ]
     coclass MyObject
     {
          [default] interface IFoo;
          interface IFoo2;
          interface IGoo;
     };
};
```

Il primo **GUID** è per la **type library**, il secondo è per il **CLSID** per l'oggetto vero e proprio. Compiliamo il file con il comando:

```
midl componente.idl
```

Inseriamo la dichiarazione di *CMyClassFactory* e di *CMyObject* nel file *MyObject.h*, e le loro implementazioni nel file *MyObject.cpp*. Scriviamo il file *componente.def* che serve per creare la DLL:

```
LIBRARY Componente.dll

DESCRIPTION ' Componente.dll '

EXPORTS   DllGetClassObject   @1PRIVATE
          DllCanUnloadNow     @2PRIVATE
          DllRegisterServer   @3PRIVATE
          DllUnregisterServer @4PRIVATE
```

Ho supposto di aver implementato anche le due funzioni **DllRegisterServer** e **DllUnregisterServer** che servono per gestire la registrazione del componente nel registro di Windows. Infatti per registrare il componente si usa il comando:

```
regsvr32 Componente.dll
```

Questo comando non fa altro che chiamare la funzione **DllRegisterServer**. Invece per deregistrarlo si usa:

```
regsvr32 /u Componente.dll
```

Anche qui si limita a chiamare la funzione **DllUnregisterServer**. Se non definiamo queste due funzioni, dobbiamo registrare manualmente il componente (con ad esempio il programma **REGEDIT.EXE**) e dovremmo inserire le seguenti chiavi:

```
HKEY_CLASSES_ROOT

  CLSID
     {BF32B161-D29B-11d2-8B40-00400559C94F} = MyObject Class

     InProcServer32=c:\\Esempi\Componente\\Debug\\Componente.
dll
```

Dove il **CLSID** è quello del file *componente.idl* (subito prima della parola chiave *coclass*), e *c:\\Esempi\Componente\Debug\Componente.dll* è ovviamente il **path** di dove si trova la DLL sul disco.

Nel file *MyObject.cpp* devo includere il file *Componente.h* generato dal **MIDL,** il file *MyObject.h* e il file *Componente_i.c* (anche questo generato dal **MIDL**) in questo ordine. Dopo queste modifiche l'inizio del file dovrebbe essere il seguente:

```
#include "Componente.h"
#include "MyObject.h"
#include "Componente_i.c"

// la Class Object
CMyClassObject g_cfMyClassObject;

// contatore degli oggetti e dei lock
ULONG g_cObjectsAndLocks = 0;

// Implementazione delle funzioni delle classi
// e delle funzioni della DLL
```

Dopo aver fatto tutto ciò si può finalmente compilare e ottenere così la DLL.

2.13 Inizializzare la libreria COM

Tutti i client e i componenti COM devono eseguire diverse operazioni in comune. Per garantire che queste operazioni comuni vangano svolte in modo standard e

compatibile, COM definisce una libreria di funzioni che implementano queste operazioni.
La libreria viene implementata in **OLE32.DLL**. Si può usare **OLE32.LIB** per effettuare il *linking* statico con la DLL.

Un processo che vuole usare COM deve richiamare la funzione **CoInitialize** per inizializzare la libreria, prima che questa consenta al processo di usare una qualsiasi delle sue funzioni (eccettuata **CoBuildVersion**, che restituisce il numero di versione). Quando il processo ha terminato di usare la libreria COM, deve richiamare **CoUninitialize**. I prototipi di queste due funzioni sono:

```
HRESULT CoInitialize(void *reserved);
// l'argomento reserved deve essere NULL

void CoUninitialize();
```

La libreria deve essere inizializzata soltanto una volta per processo. Richiamare **CoInitialize** diverse volte per processo non dà problemi, purché ad ogni chiamata di **CoInitialize** corrisponda una chiamata di **CoUninitialize**. Se **CoInitialize** è già stata richiamata da un processo, essa restituirà *S_FALSE* e non *S_OK*.
Poiché la libreria COM richiede l'inizializzazione una sola volta per processo, e poiché viene utilizzata per creare componenti, i componenti **InProc** non hanno bisogno di inizializzare la libreria. La convenzione generica è quella di gestire l'inizializzazione di COM solo negli EXE e non nelle DLL.

OLE è basato su COM, e aggiunge il supporto per le **type library**, la **clipboard**, il **drag and drop**, gli **ActiveX document**, **Automation** e i controlli **ActiveX**. La libreria **OLE** contiene il supporto addizionale per queste funzionalità. Se vogliamo utilizzare una qualunque di queste, dobbiamo richiamare **OleInitialize** e **OleUninitialize** invece di **CoInitialize** e **CoUninitialize**. Comunque le funzioni **Ole*** chiamano al loro interno le corrispondenti funzioni **Co***, quindi si possono usare direttamente le prime.

2.14 Gestione della memoria

È piuttosto frequente per una funzione in un componente, allocare un blocco di memoria, e restituire l'indirizzo al client attraverso un parametro in uscita (*[out]*). Ma chi libera questo blocco di memoria, e in che modo? Il problema maggiore è la parte relativa al chi, perché il client e il componente potrebbero essere implementati da individui diversi, essere scritti in linguaggi diversi e magari essere eseguiti in processi diversi. Occorre trovare una modalità standard per allocare e liberare la memoria.
La soluzione COM si chiama **task memory allocator**. Utilizzando il **task memory allocator**, un componente riesce a fornire al client un blocco di memoria che il client può deallocare. Quindi in COM il server alloca la memoria e il client la libera, in CORBA invece vedremo che si usa un meccanismo differente. Un vantaggio

ulteriore è che il **task memory allocator** è **thread-safe**, quindi può essere utilizzato in applicazioni multithread.
Come al solito, l'utilizzo di questo **task memory allocator** avviene tramite un'interfaccia. In questo caso l'interfaccia è **IMalloc** e viene restituita da **CoGetMalloc**. **IMalloc::Alloc** alloca un blocco di memoria, mentre **IMalloc::Free** lo libera. Tuttavia, nella maggior parte dei casi, richiamare **CoGetMalloc** per ottenere un puntatore a interfaccia, richiamare una funzione mediante il puntatore e poi rilasciare il puntatore, comporta un sacco di lavoro che in realtà non vogliamo fare. Pertanto, la libreria COM implementa alcune comode funzioni di utilità, **CoTaskMemAlloc** e **CoTaskMemFree** che gestiscono il tutto. Vediamo un esempio di utilizzo di queste due funzioni. Supponiamo di avere questa interfaccia:

```
interface IX : IUnknown
{
     HRESULT FxStringIn([in, string] wchar_t *szIn);
     HRESULT FxStringOut([out, string] wchar_t **szOut);
}
```

Dal lato server abbiamo:

```
// . . .

wchar_t stringa[256];

HRESULT Object::FxStringIn(wchar_t *szIn)
{
     wcscpy(stringa, (const wchar_t*) szIn);
     return S_OK;
}

HRESULT Object::FxStringOut(wchar_t **szOut)
{
     *szOut = (wchar_t) ::CoTaskMemAlloc(wcslen(stringa) +
1);
     if (*szOut != NULL)
     {
          wcscpy(*szOut, (const wchar_t*) stringa);
          return S_OK;
     }
     else return E_OUTOFMEM;
}

// . . .
```

Dal lato client abbiamo invece:

```
// . . .

HRESULT hr = pIX->FxStringIn(L"Stringa di prova");
assert(SUCCEDED(hr));
```

```
// ottiene una copia della stringa
wchar_t *szOut = NULL;
hr = pIX->FxStringOut(&szOut);
assert(SUCCEDED(hr));

// stampa la copia della stringa
cout << szOut << endl;

// rilascia la memoria
::CoTaskMemFree(szOut);

// . . .
```

2.15 Ereditarietà, contenimento e aggregazione

In precedenza abbiamo affermato che COM non supporta l'ereditarietà. Questo non è del tutto esatto. COM non supporta la **implementation inheritance** (l'ereditarietà dell'implementazione), ovvero quello che succede quando una classe eredita il proprio codice o la propria implementazione da una classe base. COM non supporta questo tipo di ereditarietà, ma supporta la **interface inheritance** (ereditarietà delle interfacce, solo singola però), cioè quello che succede quando una classe eredita il tipo o l'interfaccia dalla propria classe base.
Vediamo meglio questo concetto con un esempio. Supponiamo di avere la classe *CFoo* che implementa l'interfaccia *IFoo*, e di voler creare la class *CFoo2* che implementerà l'interfaccia *IFoo2* che ha le stesse funzioni di *IFoo*, più qualche altra. Allora possiamo creare l'interfaccia *IFoo2* derivandola da *IFoo*, ed aggiungendo le funzioni in più (ereditarietà delle interfacce), ma non possiamo implementare la classe *CFoo2* ereditando la classe *CFoo* (ereditarietà dell'implementazione).
COM non supporta l'ereditarietà dell'implementazione perché essa lega indissolubilmente un oggetto all'implementazione di un altro oggetto. Se cambia l'implementazione di un oggetto di base, gli oggetti derivati non funzionano più e devono essere modificati. Non è una coincidenza se lo classi di base astratte sono la forma più pura di ereditarietà dell'interfaccia, e rappresentano fra l'altro il modo di implementare le interfacce COM. I componenti COM possono essere scritti da chiunque, ovunque e in qualsiasi linguaggio. Pertanto dobbiamo essere molto fermi in merito alla protezione dei client di un componente da eventuali cambiamenti. L'ereditarietà di implementazione non garantisce il necessario livello di protezione per il client.

Per non perdere in funzionalità, COM simula l'ereditarietà dell'implementazione con il **contenimento** e l'**aggregazione**. Il **contenimento** e l'**aggregazione** sono due tecniche con le quali un componente utilizza un altro componente. Definiamo i due componenti come componente esterno (**outer)** e componente interno (**inner)**. Il componente esterno *contiene* il componente interno nel caso del **contenimento**, oppure la *aggrega* nel caso della **aggregazione**.

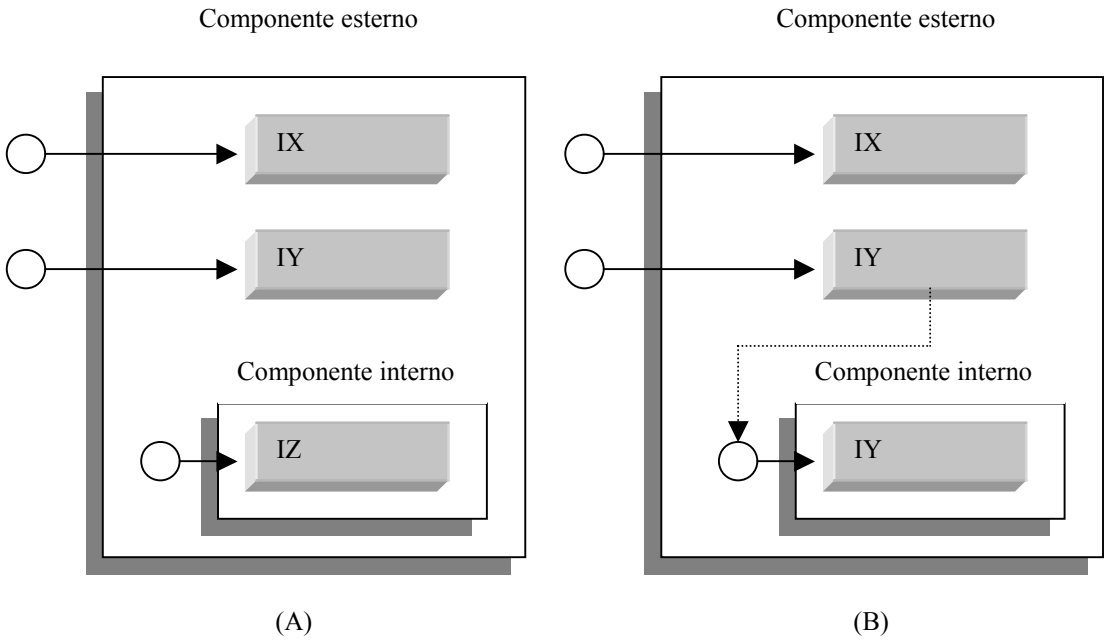


Figura 2.10 Layout interno di un componente esterno che contiene un componente interno e utilizza la sua interfaccia IZ (A). Layout interno di un componente esterno che contiene un componente interno e riutilizza la sua implementazione dell'interfaccia IY (B).

Il **contenimento** in COM è simile al contenimento in C++. Tuttavia, come ogni cosa in COM, il contenimento è realizzato a livello delle interfacce. Il componente esterno contiene dei puntatori alle interfacce del componente interno. Il componente esterno è semplicemente un client del componente interno. Il componente esterno implementa le proprie interfacce utilizzando le interfacce del componente interno (Figura 2.10a).

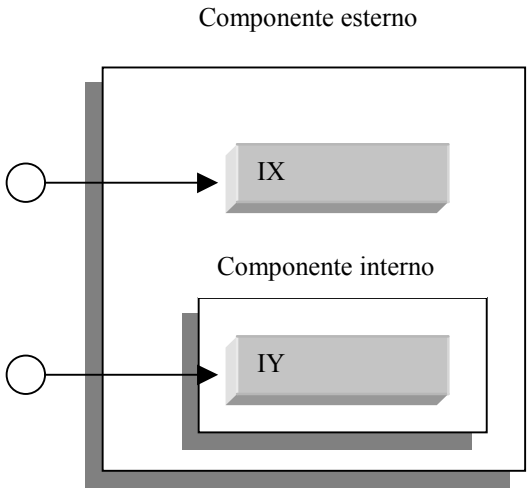


Figura 2.11. Quando il componente esterno aggrega un'interfaccia, passa il puntatore all'interfaccia direttamente al client. Non implementa nuovamente l'interfaccia per delegare le chiamate al componente interno.

Il componente esterno può anche implementare nuovamente un'interfaccia supportata dal componente interno, rimandando le chiamate al componente interno. Il componente esterno può così specializzare l'interfaccia aggiungendo una parte di codice prima o dopo la chiamata alla funzione del componente interno (vedi la Figura 2.10b).

L'**aggregazione** è una forma specializzata di contenimento. Quando un componente esterno aggrega un'interfaccia di un componente interno, non implementa nuovamente l'interfaccia e non invia in modo esplicito le chiamata al componente interno, come avviene invece per il **contenimento**. Al contrario, il componente esterno passa il puntatore all'interfaccia del componente interno direttamente al client. Il client, poi, chiama direttamente l'interfaccia che appartiene al componente interno. In questo modo, il componente esterno si risparmia di dover nuovamente implementare e *reindirizzare* tutte le funzioni di un'interfaccia (Figura 2.11).

Tuttavia, il componente esterno non è in grado di specializzare nessuna delle funzioni del componente interno. Dopo che il componente esterno ha passato l'interfaccia al client, il client colloquia direttamente con il componente interno. Il client non deve sapere che sta interagendo con due diversi componenti, altrimenti si perderebbe l'incapsulamento. Fare in modo che il componente esterno e quello interno si comportino come un unico componente è il segreto di un'**aggregazione** efficace.

2.15.1 Implementare il contenimento

Implementare il **contenimento** è altrettanto facile come usare un componente COM. Quando un componente contiene un altro componente, il client e il componente interno non devono fare praticamente nulla, anzi non sanno nemmeno che viene utilizzato il **contenimento**. L'unico che è a conoscenza del **contenimento** è il componente esterno. Vediamo le modifiche che si devono apportare facendo un esempio.
Supponiamo di voler implementare due interfacce, **IX** e **IY**, ma la prima la implementiamo direttamente, mentre la seconda la implementiamo con il **contenimento**. Il codice del componente esterno è il seguente (senza usare le macro):

```
// Componente_esterno

class CA : public IX, public IY
{
public:
     // Interfaccia IUnknown
     virtual HRESULT __stdcall
          QueryInterface(const IID& iid, void **ppv);
```

```
     virtual ULONG __stdcall Addref();
     virtual ULONG __stdcall Release();

     // Interfaccia IX
     virtual HRESULT __stdcall Fx() { cout << "Fx" << endl; }

     // Interfaccia IY
     virtual HRESULT __stdcall Fy() { m_pIY->Fy(); }

     // Costruttore e distruttore
     CA();
     ~CA();

     // Funzione di inizializzazione del componente interno
     HRESULT Init();

private:
     // Reference count
     long m_cRef;

     // Puntatore per l'interfaccia IY del componente interno
     IY* m_pIY;
};

// Membri di IUnknown
. . .
. . .


// Costruttore
CA::CA() : m_cRef(1)
{
     InterlockedIncrement(&g_cObjectsAndLocks);

     // Crea il componente interno
     HRESULT hr = Init();
}

// Distruttore
CA::~CA()
{
     InterlockedDecrement(&g_cObjectsAndLocks);

     // Rilascia il componente interno
     if (m_pIY !=NULL)
          m_pIY->Release();
}

// Inizializza il componente creando il componente interno
HRESULT CA::Init()
{
   HRESULT hr = ::CoCreateInstance(CLSID_Componente_interno,
```

```
            NULL,
            CLSCTX_INPROC_SERVER,
            IID_IY,
            (void **) &m_pIY);
    if (FAILED(hr))
            return E_FAIL
    else
            return S_OK;
}
```

Vediamo come funziona questo codice per il componente esterno. Un nuovo metodo denominato *Init* crea il componente interno nello stesso modo in cui tutti i client creano componenti, ovvero chiamando **::CoCreateInstance**. Nell'effettuare questa chiamata a funzione, il componente esterno richiede un puntatore *IY* sul componente interno, e se la chiamata ha esito positivo, esso memorizza questo puntatore in *m_pIY*.
Nel codice precedente non abbiamo mostrato le implementazioni di **QueryInterface** e delle altre funzioni di **IUnknown**. Esse funzionano esattamente come se non fosse utilizzato il **contenimento**. Quando il client chiede al componente esterno l'interfaccia **IY**, il componente esterno restituisce un puntatore alla sua interfaccia **IY**. Quando il client richiama la funzione *Fy*, il componente esterno inoltra la chiamata alla funzione *Fy* del componente interno. Questo avviene con il seguente codice:

```
virtual HRESULT __stdcall Fy() { m_pIY->Fy(); }
```

Quando il componente esterno si autodistrugge, il suo distruttore chiama **Release** sul puntatore *m_pIY*, facendo così in modo che anche il componente interno sia distrutto.
Uno dei principali utilizzi del contenimento è per estendere un'interfaccia, aggiungendo codice ad un'interfaccia già esistente in un componente contenuto. Nell'esempio precedente la modifica da fare sarebbe:

```
virtual HRESULT __stdcall CA::Fy()
{
     // codice da aggiungere prima
     . . .

     m_pIY->Fy();

     // codice da aggiungere dopo
     . . .
}
```

2.15.2 Implementare l'aggregazione

Ecco una panoramica sul funzionamento dell'aggregazione. Il client richiede al componente esterno l'interfaccia **IY**. Invece di implementare **IY**, il componente esterno richiede al componente interno la sua interfaccia **IY** e passa questo puntatore al client. Quando il client utilizza quest'interfaccia, chiama direttamente le funzioni membro di **IY** implementate dal componente interno. Il componente esterno è fuori causa per tutto quello che riguarda l'interfaccia **IY**, perché cede il controllo al componente interno.
Per quanto l'**aggregazione** sembri molto semplice, esistono alcune difficoltà nella corretta implementazione dell'interfaccia **IUnknown** per il componente interno. Vediamo perché con un esempio. Supponiamo di avere un componente aggregato. Il componente esterno supporta le interfacce **IX** e **IY**. Esso implementa l'interfaccia **IX** e aggrega l'interfaccia **IY**. Il componente interno implementa le interfacce **IY** e **IZ**. Dopo che abbiamo creato il componente esterno, otteniamo il puntatore della sua interfaccia **IUnknown**. Possiamo richiedere con esito positivo la sua interfaccia **IX** o **IY**, ma chiedendo **IZ** verrà restituito un errore di interfaccia non supportata. Se richiediamo il puntatore all'interfaccia **IY**, otteniamo il puntatore all'interfaccia **IY** del componente interno. Se richiediamo **IZ** da questo puntatore, la cosa avrà successo. Questo perché il componente interno implementa le funzioni di **IUnknown** per l'interfaccia **IY**. Ma se richiediamo l'interfaccia **IX** dal puntatore a **IY**, otterremo un errore perché il componente interno non supporta **IX**. Questo comportamento non va bene perché costringe il client a sapere come sono implementate le interfacce.
Il problema nasce dall'interfaccia **IUnknown** del componente interno. Il client ne vede due, quella del componente esterno e quella del componente interno. Questo confonde il client perché ogni **IUnknown** implementa una diversa **QueryInterface** e ogni **QueryInterface** supporta un diverso insieme di interfacce. Il client dovrebbe essere completamente indipendente dall'implementazione del componente aggregato. Non dovrebbe sapere che il componente esterno sta aggregando il componente interno e non dovrebbe mai vedere l'**IUnknown** del componente interno. Pertanto dobbiamo nascondere al client l'**IUnknown** del componente interno, e fornirgli sempre quella del componente esterno, che è chiamata anche **Outer IUnknown** o **Controlling IUnknown**.
Il modo più semplice per il componente interno di usare la **IUnknown** esterna, è quello di *reindirizzare* le chiamate verso quest'ultima. Per fare ciò al componente interno serve un puntatore, e deve inoltre sapere che sta per essere aggregato. Negli esempi precedenti abbiamo visto che a **CoCreateInstance** e **IClassFactory::CreateInstance** viene passato un puntatore a **IUnknown** che non abbiamo utilizzato:

```
HRESULT __stdcall CoCreateInstance(
     const CLSID& clsid,
     IUnknown* pUnknownOuter, // componente esterno
     DWORD dwclsContext, // contesto del server
     const IID& iid,
     void **ppv);

HRESULT __stdcall CreateInstance(
     IUnknown* pUnknownOuter, // componente esterno
```

```
const IID& iid,
void **ppv);
```

Il componente esterno passa il puntatore alla sua interfaccia **IUnknown** al componente interno utilizzando il parametro *pUnknownOuter*. Se il puntatore è diverso da *NULL*, il componente viene aggregato. Utilizzando questo puntatore il componente interno non solo sa di essere aggregato, ma sa anche chi lo aggrega. Se un componente non viene aggregato (*pUnknownOuter* uguale a *NULL*), utilizza la propria implementazione di **IUnknown**, se invece è aggregato delega la chiamate alla **IUnknown** esterna.

Per supportare l'aggregazione, in realtà il componente interno implementa due interfacce **IUnknown**, una **non delegante** e una **delegante**. La **nondelegating IUnknown** implementa l'interfaccia **IUnknown** nel modo consueto, mentre la **delegating IUnknown** può rimandare le chiamate alle funzioni membro di **IUnknown**, sia alla **IUnknown** esterna, sia alla **IUnknown** non delegante. Se il componente interno non è aggregato, la **IUnknown** delegante invia le chiamate alla **IUnknown** non delegante. Se il componente interno è aggregato, la **IUnknown** delegante invia le chiamate alla **IUnknown** esterna, che viene implementata dal componente esterno. I client dell'aggregato chiamano la **IUnknown** delegante, mentre il componente esterno manipola il componente interno attraverso la **IUnknown** non delegante. Queste due situazioni sono illustrate rispettivamente nella Figura 2.12 e nella Figura 2.13. Nella prima, possiamo vedere che la parte **IUnknown** di **IY** richiama l'implementazione della **IUnknown** delegante. Questa richiama a sua volta l'**IUnknown** non delegante che è implementata nel modo consueto. Nella seconda invece possiamo vedere un componente che aggrega **IY**. Quando è aggregata, la **IUnknown** delegante richiama la **IUnknown** implementata nel componente esterno.

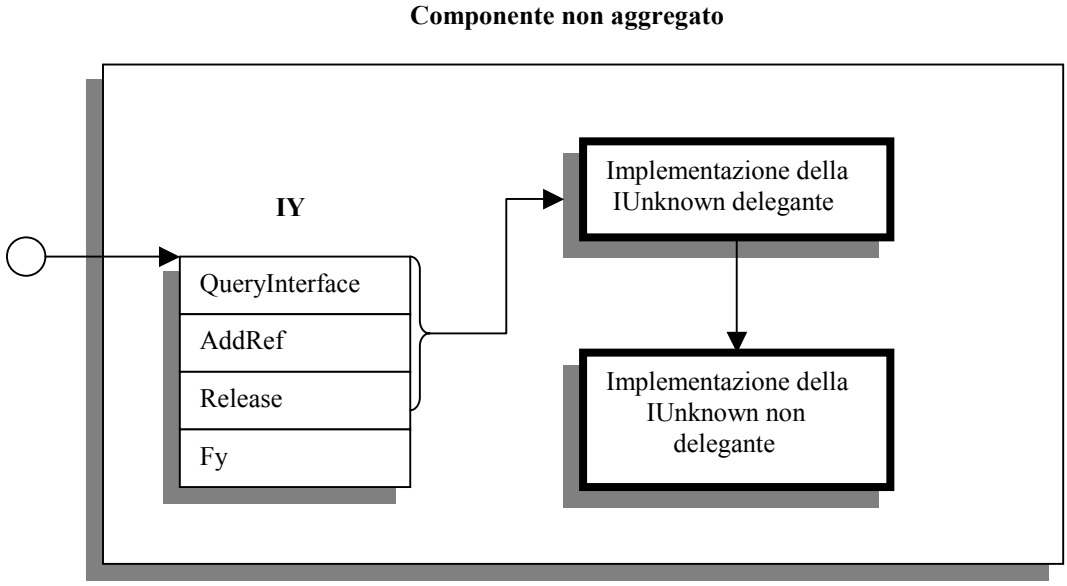


Figura 2.12. Quando il componente non è aggregato, la sua **IUnknown** delegante inoltra le chiamate alla **IUnknown** non delegante.

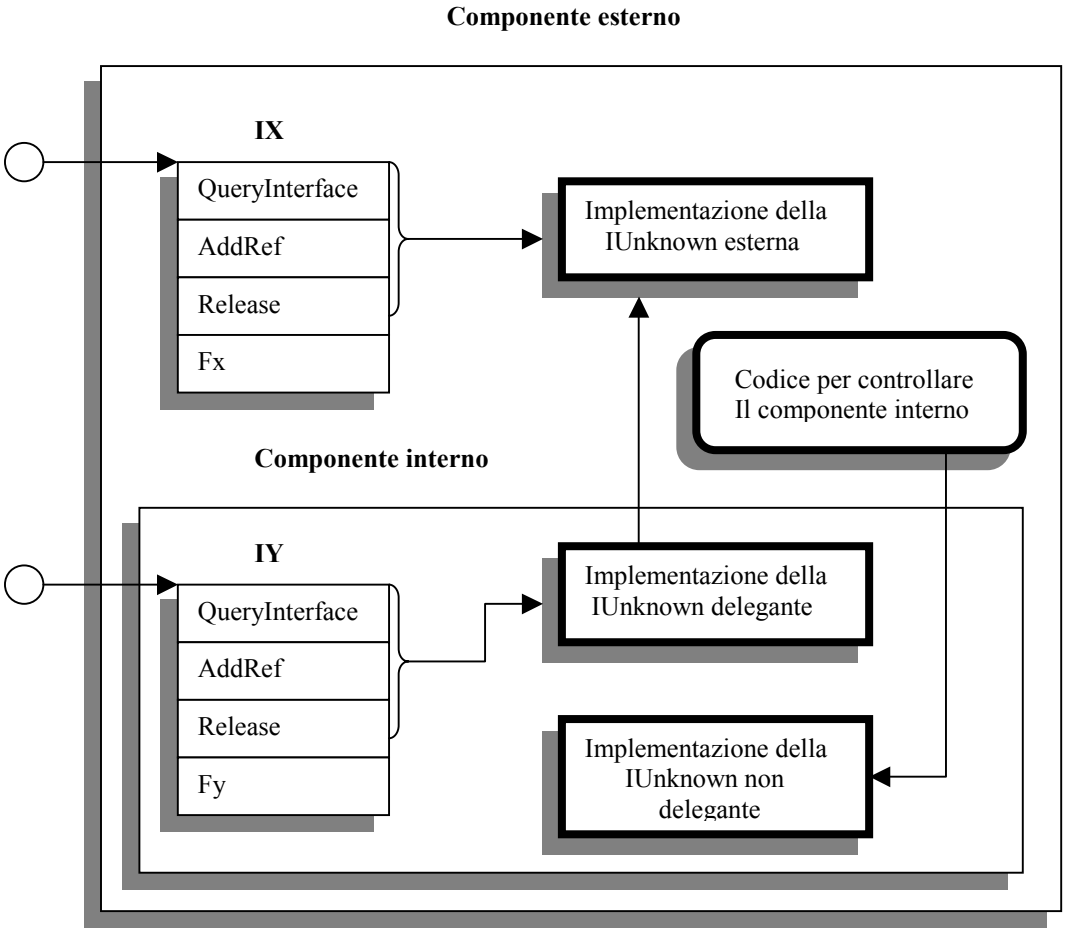


Figura 2.13. Quando il componente è aggregato, la sua **IUnknown** delegante invia le chiamate alla **IUnknown** esterna.

Il componente esterno richiama la **IUnknown** non delegante per controllare la durata del componente interno.

Pertanto, ogni volta che un componente richiama un membro di **IUnknown** sull'interfaccia **IY**, esso sta chiamando la **IUnknown** delegante che invia la chiamata alla **IUnknown** esterna. Il risultato è che il componente interno sta ora utilizzando l'implementazione della **IUnknown** del componente esterno.

Come abbiamo visto nel componente che supporta l'**aggregazione** dobbiamo avere due diverse implementazioni di **IUnknown**, ma il C++ non consente di avere due implementazioni della stessa interfaccia in una singola classe. Pertanto dobbiamo modificare il nome di una delle due **IUnknown** in modo che i loro nomi non entrino in contrasto. Usiamo ad esempio il nome **INondelegatingUnknown**, tenendo presente che a COM non interessano i nomi delle interfacce, ma solamente il layout della **vtable**. Vediamo la dichiarazione di questa nuova interfaccia:

```
struct INondelegatingUnknown
{
     virtual HRESULT __stdcall
        NondelegatingQueryInterface(const IID& iid,
           void** ppv) = 0;
     virtual ULONG __stdcall NondelegatingAddRef() = 0;
```

```
     virtual ULONG __stdcall NondelegatingRelease() = 0;
};
```

Queste funzioni vengono implementate esattamente allo stesso modo di come abbiamo precedentemente implementato **AddRef** e **Release** per **IUnknown**. Tuttavia apportiamo un minimo, ma fondamentale cambiamento, nell'implementazione della funzione *NondelegatingQueryInterface* (nella classe *CB* che implementa il componente interno):

```
HRESULT __stdcall CB::NondelegatingQueryInterface(
          const IID& iid, void** ppv)
{
     if (iid == IID_IUnknown)
          *ppv = static_cast<INondelegatingUnknown*>(this);
     else if (iid == IID_IY)
          *ppv = static_cast<IY*>(this);
     else
     {
          *ppv = NULL;
          return E_NOINTERFACE;
     }

     reinterpret_cast<IUnknown*>(*ppv)->AddRef();
     return S_OK;
}
```

È da notare il cast del puntatore *this* del componente interno a un puntatore del tipo **INondelegatingUnknown**. Questo cast è molto importante. Con il primo cast del puntatore *this* a **INondelegatingUnknown**, garantiamo di restituire sempre la **IUnknown** non delegante. Questa restituisce sempre un puntatore a se stessa, quando le viene richiesta **IID_IUnknown**. Senza questo cast, verrebbe invece restituita la **IUnknown** delegante. Quando il componente è aggregato, la **IUnknown** delegante dirotta tutte le chiamate **QueryInterface**, **AddRef** e **Release** sull'oggetto esterno.
I client del componente aggregato non ottengono mai puntatori per la **IUnknown** non delegante del componente interno. Ogni volta che il client richiede un puntatore a **IUnknown**, esso ottiene sempre un puntatore alla **IUnknown** del componente esterno. Soltanto il componente esterno ottiene un puntatore per la **IUnknown** non delegante del componente interno.
Fortunatamente l'implementazione della **IUnknown** delegante è molto facile. Essa invia le chiamate o alla **IUnknown** esterna o a quella non delegante. Di seguito vediamo la dichiarazione di un componente che supporta l'**aggregazione** (*CB*). Il componente contiene un puntatore denominato *m_pUnknownOuter*. Quando il componente è aggregato, questo puntatore punta alla **IUnknown** esterna. Se il componente non è aggregato, questo puntatore punta invece alla **IUnknown** non delegante. Ogni qualvolta viene chiamata la **IUnknown** delegante, la chiamata viene dirottata sull'interfaccia a cui punta *m_pUnknownOuter*. La **IUnknown** viene implementata *inline*:

```
class CB : public IY,
     public INondelegatingUnknown
{
public:
     // IUnknown delegante
     virtual HRESULT __stdcall
          QueryInterface(const IID& iid, void **ppv)
     {
          // delega QueryInterface
          return m_pUnknownOuter->QueryInterface(iid, ppv);
     }
     virtual ULONG __stdcall AddRef()
     {
          // delega AddRef
          return m_pUnknownOuter->AddRef();
     }
     virtual ULONG __stdcall Release()
     {
          // delega Release
          return m_pUnknownOuter->Release();
     }

     // IUnknown non delegante
     virtual HRESULT __stdcall
          NondelegatingQueryInterface(const IID& iid,
             void **ppv);
     virtual ULONG __stdcall NondelegatingAddRef();
     virtual ULONG __stdcall NondelegatingRelease();

     // interfaccia IY
     virtual HRESULT __stdcall Fy() { cout << "Fy" << endl; }

     // costruttore
     CB(IUnknown* m_pUnknownOuter);

     // distruttore
     ~CB();

private:
     long m_cRef;
     IUnknown* m_pUnknownOuter;
};
```

Vediamo come viene creato il componente interno analizzando tre funzioni: la funzione *Init* del componente esterno, che avvia la procedura di creazione; la funzione **CreateInstance** della **Class Factory** del componente interno; il costruttore del componente interno.

Il primo passo che il componente esterno compie quando aggrega un componente è quello di creare il componente interno. La differenza principale tra **aggregazione** e **contenimento** è che nella prima il componente esterno passa un puntatore alla sua

interfaccia **IUnknown** al componente interno. Il frammento di codice sottostante illustra il modo in cui il componente esterno crea il componente interno. Il secondo parametro di **CoCreateInstance** è un puntatore all'interfaccia **IUnknown** del componente esterno, mentre il quarto richiede un puntatore all'interfaccia **IUnknown** del componente interno. La **Class Factory** restituirà un puntatore alla **IUnknown** non delegante del componente interno. Come abbiamo già visto, il componente esterno ha bisogno di questo puntatore per chiamare **QueryInterface** sul componente interno. Il componente esterno deve richiedere il puntatore a **IUnknown** in questo punto, altrimenti non lo potrà più ottenere; deve inoltre memorizzare il puntatore alla **IUnknown** non delegante del componente interno, per poterla utilizzare in un secondo tempo (la variabile *m_pUnknownInner*). In questo esempio non abbiamo bisogno di fare il cast esplicito del puntatore *this* ad un puntatore **IUnknown**, perché *CA* eredita soltanto da **IX** e, pertanto, il cast implicito non crea ambiguità:

```
HRESULT CA::Init()
{
     IUnknown* pUnknownOuter = this; // cast implicito
     HRESULT hr = CoCreateInstance(CLSID_Componente_interno,
          pUnknownOuter,
          CLSCTX_INPROC_SERVER,
          IID_IUnknown,
          (void **) &m_pUnknownInner);
     if (FAILED(hr))
          return E_FAIL;
     else
          return S_OK;
}
```

Questa funzione può essere chiamata nel costruttore del componente esterno, oppure nella sua funzione **IClassFactory::CreateInstance** (funzione che a parte questo rimane invariata).

L'implementazione di **IClassFactory::CreateInstance** del componente interno viene modificata in modo da utilizzare **INondelegatingUnknown** invece di **IUnknown**. Una cosa da notare è che se l'*iid* è diverso da *IID_IUnknown* e si vuole aggregare il componente, la funzione **CreateInstance** da un esito negativo. Infatti il componente interno può restituire soltanto un'interfaccia **IUnknown** quando viene aggregato, poiché il componente esterno non può ottenere il puntatore alla **IUnknown** non delegante in nessun altro momento, in quanto le chiamate di **QueryInterface** verranno delegate alla **IUnknown** esterna:

```
HRESULT __stdcall CBFactory::CreateInstance(
          IUnknown* pUnknownOuter,
          const IID& iid,
          void **ppv)
{
     // per aggregare, iid deve essere IID_IUnknown
     if ((pUnknownOuter != NULL) && (iid != IID_IUnknown))
```

```
            return CLASS_E_NOAGGREGATION;

        // crea il componente
        CB* pB = new CB(pUnknownOuter);

        if (pB == NULL)
            return E_OUTOFMEMORY;

        // ottiene l'interfaccia richiesta
        HRESULT hr = pB->NondelegatingQueryInterface(iid, ppv);
        pB->NondelegatingRelease();

        return hr;
}
```

Questa funzione chiama **NondelegatingQueryInterface**, non **QueryInterface**, per ottenere l'interfaccia richiesta sul componente interno appena creato. Infatti se il componente interno sta per essere aggregato, delegherebbe una chiamata a **QueryInterface** alla **IUnknown** esterna. La **Class Factory** deve restituire un puntatore alla **IUnknown** non delegante, quindi richiama **NondelegatingQueryInterface**.

Il costruttore del componente interno inizializza *m_pIUnknownOuter*, che viene utilizzato dalla **IUnknown** delegante per reindirizzare le chiamate sia sull'implementazione non delegante, sia sull'implementazione della **IUnknown** esterna. Se il componente non viene aggregato (*pUnknownOuter* è *NULL*), il costruttore imposta *m_pUnknownOuter* sulla **IUnknown** non delegante. Questo viene descritto dal codice sottostante:

```
CB::CB(IUnknown* pUnknownOuter)
        : m_cRef(1)
{
        InterlockedIncrement(&g_cObjectsAndLocks);

        if (pUnknownOuter == NULL)
        {
        // non viene aggregato, utilizza
        // la IUnknown non delegante
            m_pUnknownOuter = reinterpret_cast<IUnknown*>(
                static_cast<INondelegatingUnknown*>(this));
        }
        else
        {
            // viene aggregato, utilizza la IUnknown esterna
            m_pUnknownOuter = pUnknownOuter;
        }
}
```

Vediamo adesso gli altri cambiamenti necessari nel componente esterno (ricordiamo che implementa l'interfaccia **IX** e offre l'interfaccia **IY** attraverso l'**aggregazione**):

```
class CA : public IX // non derivo da IY
{
public:
     // IUnknown
     virtual HRESULT __stdcall
          QueryInterface(const IID& iid, void **ppv);
     virtual ULONG __stdcall AddRef();
     virtual ULONG __stdcall Release();

     // interfaccia IX
     virtual HRESULT __stdcall Fx() { cout << "Fx" << endl; }

     // costruttore
     CA();

     // distruttore
     ~CA();

     // funzione di inizializzazione del componente interno
     HRESULT Init();

private:
     // reference count
     long m_cRef;

     // puntatore alla IUnknown del componente interno
     IUnknown* m_pUnknownInner;
};
```

Il componente dichiarato in questo codice non sembra supportare l'interfaccia **IY**: esso non eredita da **IY** e non ne implementa alcun membro. Il componente esterno utilizza l'implementazione di **IY** del componente interno. Questo si vede nella funzione **QueryInterface** che restituisce un puntatore ad un'interfaccia dell'oggetto interno. Nel frammento di codice sottostante, la variabile membro *m_pIUnknownInner* contiene l'indirizzo della **IUnknown** del componente interno (come abbiamo visto viene inizializzato nella funzione **Init** del componente esterno):

```
HRESULT __stdcall CA::QueryInterface(
          const IID& iid, void **ppv)
{
     if (iid == IID_IUnknown)
          *ppv = static_cast<IX*>(this);
     else if (iid == IID_IX)
          *ppv = static_cast<IX*>(this);
     else if (iid == IID_IY)
          return m_pUnknownInner->QueryInterface(iid, ppv);
     else
     {
          *ppv = NULL;
```

```
            return E_NOINTERFACE;
        }

        reinterpret_cast<IUnknown*>(*ppv)->AddRef();
        return S_OK;
}
```

Se volessimo rimanere ancora più distaccati dal componente interno per prevedere, ad esempio, l'utilizzo di interfacce future aggiunte al componente, dovremmo eliminare il controllo sull'interfaccia **IY**; cioè eliminando il controllo *if (iid == IID_IY)* possiamo utilizzare anche le eventuali interfacce future del componente interno (si ottiene così la cosiddetta **aggregazione alla cieca**).

2.16 Interfacce IDispatch e Automation

Fino a questo punto, per far comunicare il client con il componente, abbiamo usato sempre un'interfaccia COM, ma esistono anche modi diversi per permettere questa comunicazione. Ad esempio un altro modo per pilotare il componente è **Automation** (una volta si chiamava **OLE Automation**).
Questo metodo viene utilizzato da applicazioni quali **Microsoft Word** e **Microsoft Exel**, da linguaggi interpretati come **Visual Basic** e **Java** (che possono utilizzare anche il metodo standard), e dai linguaggi di scripting.
Automation rende più facile l'accesso ai componenti COM da parte di questi linguaggi, e inoltre ne semplifica la scrittura. **Automation** si concentra sulla verifica del tipo di un oggetto in fase di esecuzione, a scapito della velocità e della verifica in fase di compilazione. Tuttavia, anche se **Automation** è più semplice per chi scrive macro, richiede molto più impegno da parte dello sviluppatore C++. In molti sensi, **Automation** sostituisce il codice generato dal compilatore con il codice scritto dallo sviluppatore.
Automation non è separata da COM, ma è costruita su di esso: un **server Automation** è un componente COM che implementa l'interfaccia standard **IDispatch**. Un **controller Automation** (così si chiama il client) è un client COM che comunica con il **server Automation** attraverso la sua interfaccia **IDispatch**. Il client non richiama direttamente le funzioni implementate dal server. Al contrario il client utilizza funzioni membro dell'interfaccia **IDispatch**, per richiamare indirettamente le funzioni del componente.
Quasi tutti i servizi che possono essere forniti attraverso interfacce COM, possono anche essere serviti attraverso **IDispatch**. **IDispatch** offre un metodo alternativo per far comunicare il client con il componente. Invece di fornire tante interfacce personalizzate specifiche per i servizi che offre, un componente può offrire quei servizi attraverso un'unica interfaccia standard, appunto **IDispatch**. Quello che fa questa interfaccia è abbastanza semplice, prende il nome di una funzione e la esegue.
La dichiarazione di **IDispatch**, tratta dal file **oaidl.idl**, è la seguente:

```
interface IDispatch : IUnknown
```

```
{
    HRESULT GetTypeInfoCount([out] UINT* pctinfo);

    HRESULT GetTypeInfo([in] UINT iTInfo,
              [in] LCID lcid,
              [out] ItypeInfo** ppTInfo);

    HRESULT GetIDsOfNames([in] const IID& riid,
              [in, size_is(cNames)] LPOLESTR* rgszNames,
              [in] UINT cNames,
              [in] LCID lcid,
              [out, size_is(cNames)] DISPID* rgDispId);

    HRESULT Invoke([in] DISPID dispIdMember,
              [in] const IID& riid,
              [in] LCID lcid,
              [in] WORD wFlags,
              [in, out] DISPPARAMS* pDispParams,
              [out] VARIANT* pVarResult,
              [out] EXCEPINFO* pExcepInfo,
              [out] UINT* puArgErr);
```

Le due funzioni più interessanti di **IDispatch** sono **GetIDsOfNames** e **Invoke**. La prima legge il nome di una funzione e restituisce il suo **dispatch ID**, o **DISPID**. Un **DISPID** non è un **GUID**, ma soltanto un intero (*long* oppure *LONG*). Il **DISPID** identifica una funzione, ma non è univoco, se non in una specifica implementazione di **IDispatch**. Ogni funzione di un'interfaccia che eredita da **IDispatch** riceve il proprio **IID** (che è definito anche **DIID** ovvero **ID** dell'interfaccia **IDispatch**) nella sua definizione in **IDL**.
Per eseguire la funzione, l'**Automation controller** passa il **DISPID** alla funzione membro **Invoke**, la quale può utilizzarlo come indice in un array di puntatori a funzioni, molto simile alle normali interfacce COM. Tuttavia, l'**Automation server** non deve necessariamente implementare **Invoke** in questo modo. Un semplice **Automation server** potrà utilizzare un'istruzione *case* che esegue un codice diverso a seconda del **DISPID**.
Il modo in cui funziona **Invoke** ricorda anche il modo in cui funziona una *vtable*, dietro le quinte. Infatti implementa un insieme di funzioni alle quali si accede tramite un indice. Una *vtable* è un array di puntatori a funzione ai quali accede tramite un indice. Mentre le *vtable* funzionano automaticamente grazie al compilatore C++, **Invoke** funziona grazie al duro lavoro del programmatore. Tuttavia, le *vtable* in C++ sono statiche, e il compilatore funziona soltanto in fase di compilazione. Se il programmatore vuole creare una *vtable* in fase di esecuzione, partendo da zero, è lasciato completamente a se stesso. Invece è molto facile creare un'implementazione generica di **Invoke**, che può essere personalizzata al volo, per implementare servizi diversi.
Non ci occuperemo dell'implementazione di **IDispatch** perché è un lavoro complicato e ripetitivo, inoltre esistono molte librerie di classi (tra cui l'**Active Template Library** o **ATL** di cui vedremo alcuni esempi) che automatizzano il tutto.

2.16.1 Dispinterface

Un'implementazione di **IDispatch::Invoke** condivide un'altra somiglianza con la *vtable*: entrambe definiscono interfacce. L'insieme di funzioni fornite da un'implementazione di **IDispatch::Invoke** si chiama **dispatch interface** o in breve **dispinterface**. La definizione di un'interfaccia COM è un puntatore a un array di puntatori a funzione, dove le prime tre funzioni sono **QueryInterface**, **AddRef** e **Release**. Una definizione più generica di un'interfaccia è un insieme di funzioni e di variabili, attraverso le quali due parti di un programma possono comunicare. Un'implementazione di **Invoke** costituisce un insieme di funzioni attraverso le quali comunicano un **Automation controller** e un **Automation server**. Pertanto non è difficile vedere che le funzioni implementate da **Invoke** costituiscono un'interfaccia, ma non un'interfaccia COM. La Figura 2.14 è una rappresentazione grafica di una **dispinterface**.

A sinistra della figura abbiamo un'interfaccia COM tradizionale, **IDispatch**, implementata per mezzo di una *vtable*. A destra troviamo una **dispinterface**. La figura illustra una possibile implementazione di **Invoke** e di **GetIDsOfNames**: un array di nomi di funzioni e un array di puntatori a funzione indicizzati dai **DISPID**. Questa è soltanto una delle soluzioni possibili. Per **dispinterface** di maggiori dimensioni, **GetIDsOfNames** risulta più veloce se il nome che viene passato viene usato come chiave in una tabella hash. Naturalmente è anche possibile utilizzare un'interfaccia COM per implementare **Invoke** (Figura 2.15).

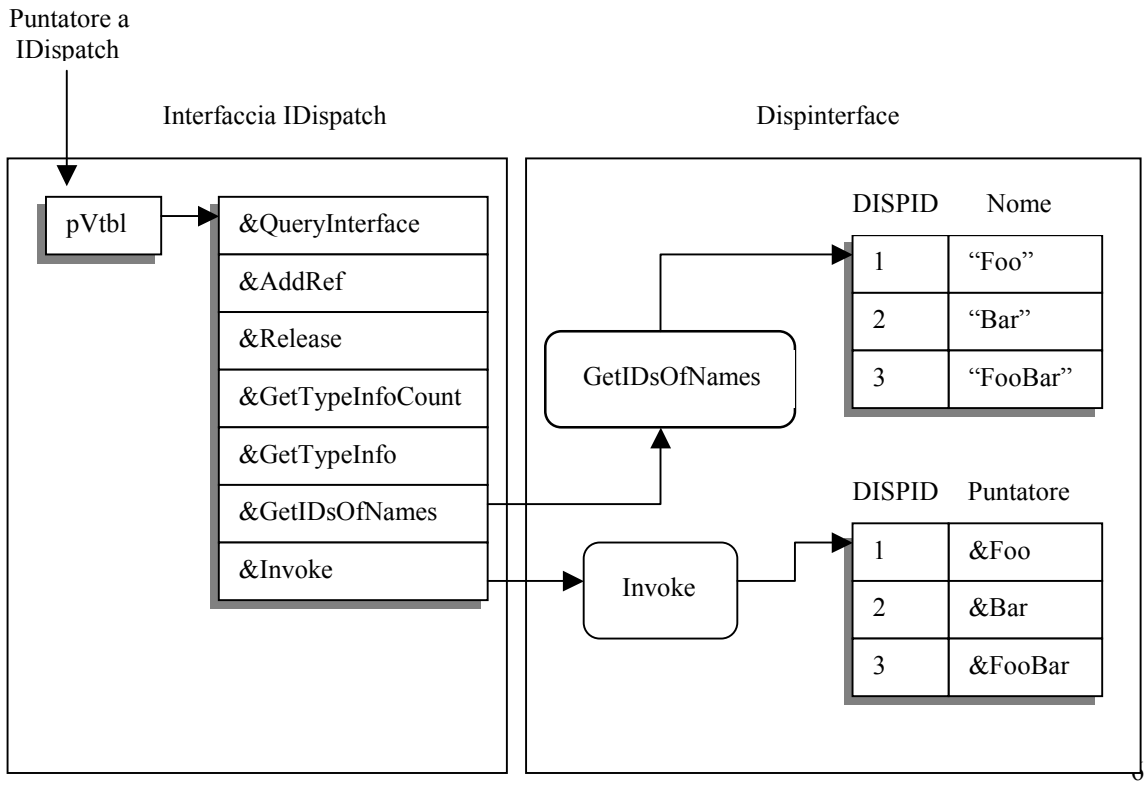


Figura 2.14 Le **dispinterface** vengono implementate da **IDispatch** e sono distinte dalle interfacce COM. Questa figura rappresenta soltanto una delle possibili implementazioni di **IDispatch::Invoke**.

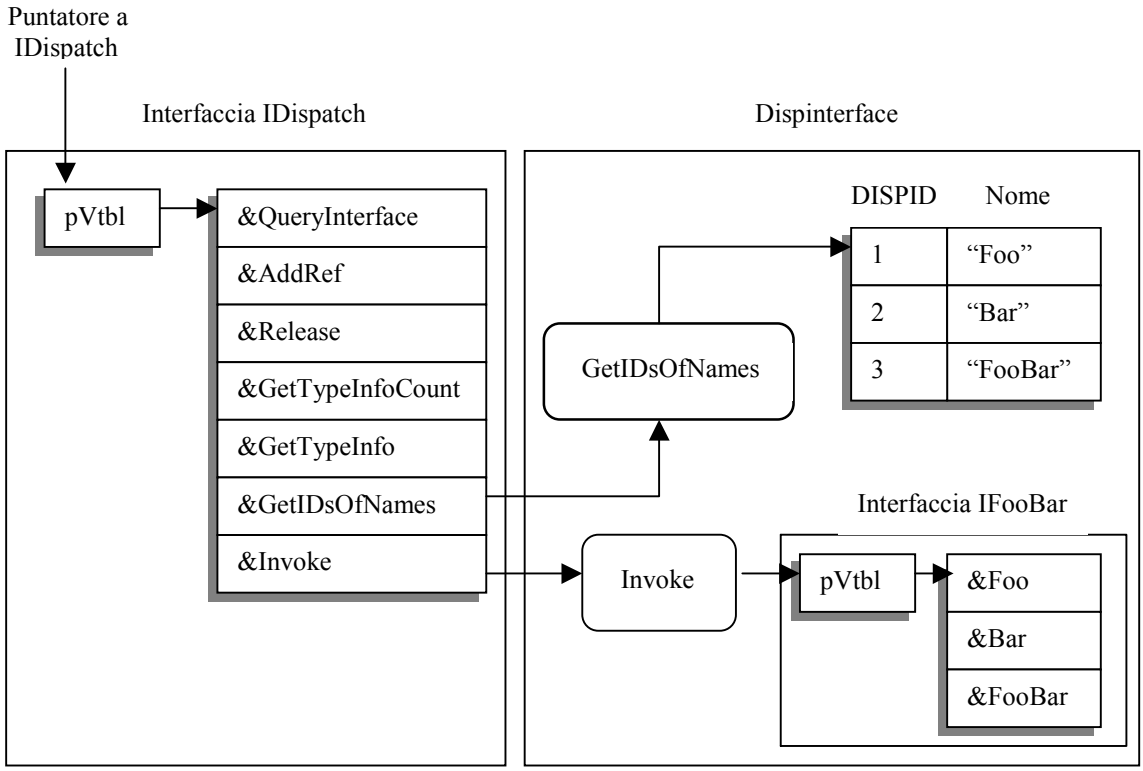
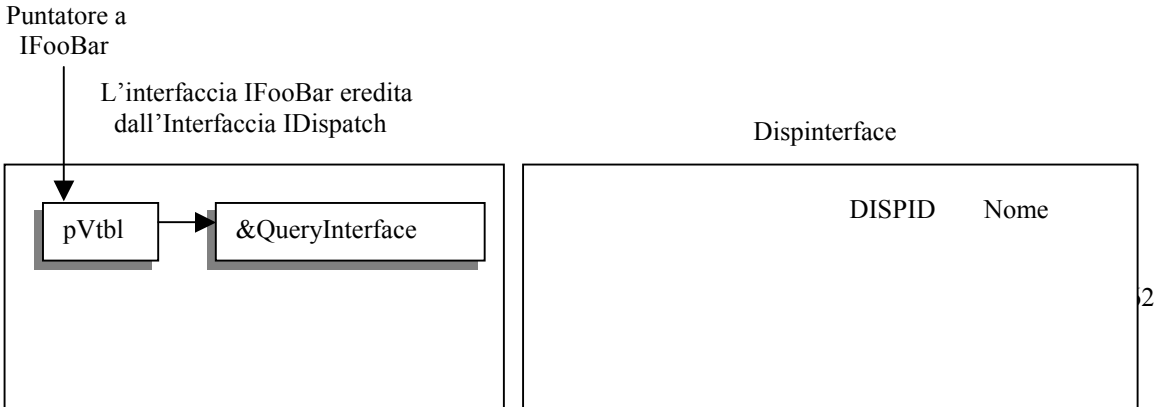


Figura 2.15. Implementazione di **IDispatch::Invoke** usando un'interfaccia COM.

2.16.2 Dual interface

La Figura 2.15 non è l'unico modo per implementare una **dispinterface** utilizzando un'interfaccia COM. Un altro metodo, illustrato nella Figura 2.16, è quello di fare in modo che l'interfaccia COM che implementa **IDispatch::Invoke** erediti da **IDispatch** e non da **IUnknown**. Questo è un modo per implementare un tipo di interfaccia conosciuto come **dual interface**, cioè interfaccia duale. Una **dual interface** è una **dispinterface** che rende direttamente disponibili tramite *vtable*, tutte le funzioni già disponibili attraverso **Invoke**.



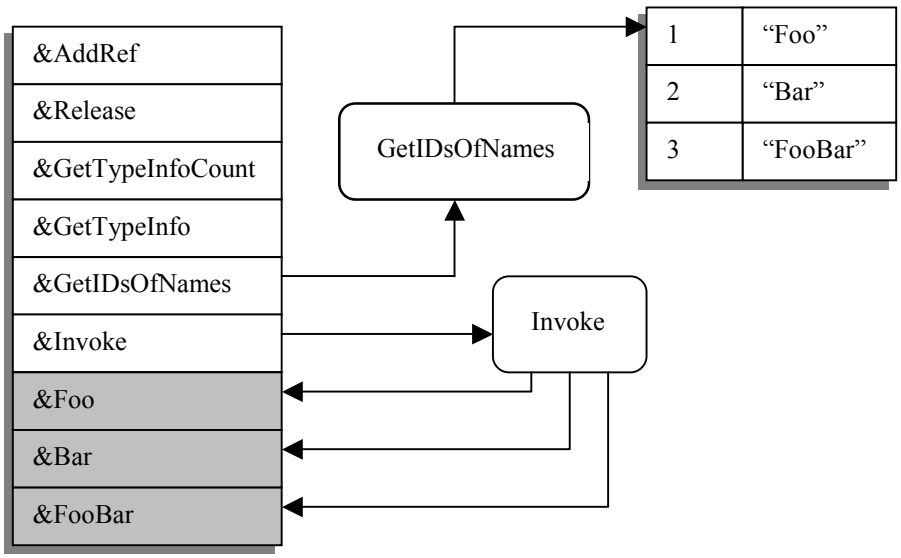


Figura 2.16 Una **dual interface** è un'interfaccia COM che eredita da **IDispatch**. I membri dell'interfaccia sono accessibili sia attraverso **Invoke**, sia attraverso la *vtable*.

Le **dual interface** sono il metodo preferito per implementare le **dispinterface**, infatti consentono ai programmatori C++ di effettuare le proprie chiamate attraverso la *vtable*: tali chiamate non soltanto sono più facili da implementare per i programmatori C++, ma vengono anche eseguite più rapidamente. Le macro e i linguaggi interpretati possono anche essi utilizzare i servizi dei componenti che implementano le **dual interface**. Invece di effettuare le chiamate attraverso la *vtable*, possono farlo mediante il metodo **Invoke**. Un programma **Visual Basic** può collegarsi sia alla parte **dispinterface**, sia alla parte *vtable* della **dual interface**. Se dichiariamo una variabile in **Visual Basic** come di tipo *Object*, si collega tramite la **dispinterface**:

```
Dim doc As Object  ' collegamento tramite dispinterface
Set doc = Application.ActiveDocument
doc.Activate
```

Se però diamo alla variabile il tipo dell'oggetto, **Visual Basic** effettua le chiamate tramite la *vtable*:

```
Dim doc As Document  ' collegamento tramite vtable
Set doc = Application.ActiveDocument
doc.Activate
```

Avendo specificato il tipo dell'oggetto, che in questo caso è *Document*, diamo la possibilità alla parte di **Visual Basic** che si occupa di **Automation** di accedere alle informazioni sull'oggetto e quindi di usarlo come un oggetto COM tradizionale.

2.17 Un server ATL con Visual C++

Scrivere un componente COM non è una cosa facilissima. Ci sono molte complicazioni sulla sua creazione, sul **reference counting**, ecc. che fanno perdere molto tempo ai programmatori. Siccome queste operazioni si implementano quasi sempre in modo standard, molti produttori di tool di sviluppo hanno aggiunto delle funzioni ai loro prodotti, non solo per supportare COM, ma per semplificare al massimo il processo di creazione dei componenti, lasciando così al programmatore solo il compito di implementare le interfacce custom. Un esempio è la libreria **ATL** (**Active Template Library**) di Microsoft.

L'**ATL** fu progettata originariamente come un metodo per scrivere componenti COM piccoli e veloci. Nella sua prima versione, non aveva nessuna funzione per l'uso di componenti visuali (tipo controlli **ActiveX**) ed era limitata ai componenti senza interfaccia utente. Ma già dalla versione 2.0 sono stati aggiunti dei **template** necessari per costruire controlli **ActiveX**, e per il collegamento ai database. Nella versione attuale è il metodo migliore per scrivere componenti COM (più efficienti) in C++ (superiore anche alla **MFC**) ed è stata inclusa dalla Microsoft nel **Visual C++**.

Vediamo come creare il componente visto in precedenza con il **Visual C++** (versione 5 o superiore).

Nella Figura 2.17 è schematizzato il componente con le sue interfacce. Ricordiamo che l'interfaccia **IFoo2** è derivata da **IFoo**, mentre **IGoo** è completamente indipendente. Genereremo questo componente in modo da essere **single-threaded**, cioè può essere usato da un singolo **thread**, e con il supporto della interfaccia **duale** (**dual**) per fare in modo che possa essere usato da qualsiasi client (compresi quelli scripting).
Poiché un modulo (una DLL o un EXE) può implementare più di un componente, il **Wizard** dell'**ATL** (un generatore automatico di codice fornito con il **Visual C++**) ferma il processo di creazione del componente dopo due passi. Il primo crea il modulo, mentre il secondo aggiunge i componenti.

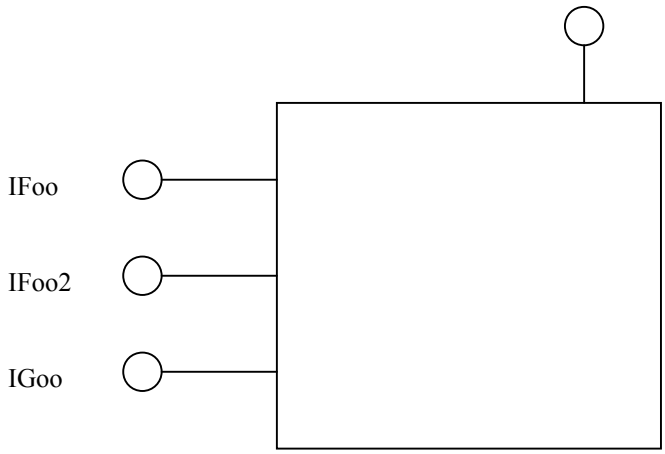


Figura 2.17 Schema del componente e delle sue interfacce che implementaremo usando il **Visual C++** e la libreria **ATL**.

La creazione del modulo è molto semplice. Basta selezionare **New…** dal menù **File** e quindi selezionare la voce **Project**, riempire la casella **Project name** con il nome del progetto, nel nostro caso "*MyObjectMod*", e premere il tasto **OK**. A questo punto entriamo nel **Wizard ATL**. La scelta principale qui è il tipo di modulo, DLL (per i server **InProc**), EXE (per i server **OutOfProc**), o **NT Service** EXE. Selezioniamo la DLL. Ignoriamo i successivi tre **check box** per il momento, perché non vogliamo usare né **MFC**, né **MTS**. Siccome il server è **InProc** non abbiamo bisogno nemmeno del codice **proxy/stub**. Dopo aver premuto il tasto **Finish**, otteniamo un progetto che contiene un gruppo di file, questi sono:

- Il file **MyObjectMod.idl** per il progetto che contiene solo la dichiarazione del blocco per la **Type Library**.
- Un file **.def** per il linker che esporta le quattro funzioni che le DLL COM devono esportare (come abbiamo visto in precedenza).
- Un file **.rc** che contiene la versione e la stringa per il nome del progetto.
- Un file *header* che contiene la definizione degli **ID**.
- I file **stdafx.h** e **stdafx.cpp** per l'inclusione degli *header* precompilati.
- Il file **MyObjectMod.cpp** che contiene l'implementazione per tutte le funzioni globali necessarie per una DLL COM.

Vediamo l'ultimo file in dettaglio:

```
#include "stdafx.h"
#include "resource.h"
#include <initguid.h>  // dichiara la struttura GUID
#include "MyObjectMod.h"  // generato da MIDL
#include "MyObjectMod_i.c"  // generato da MIDL

CcomModule _Module;

BEGIN_OBJECT_MAP(ObjectMap)
END_OBJECT_MAP()
```

Il file *initguid.h* è un file standard di **OLE** che si deve includere esattamente una volta nel progetto; definisce la struttura per i GUID.
I file *MyObjectMod.h* e *MyObjectMod_i.c* sono generati dal compilatore **MIDL** dal file *MyObjectMod.idl* quando si compila il progetto. Come abbiamo visto in precedenza, il primo contiene la dichiarazione delle interfacce e dei componenti, il secondo definisce i **GUID** che stiamo usando.
Successivamente troviamo la dichiarazione della variabile globale *_Module*. Questo oggetto contiene la **Class Factory**, più altro codice comune ai componenti come ad esempio il codice necessario per la registrazione.
Infine troviamo la dichiarazione della mappa dei componenti che conterrà il modulo. Questa mappa sarà riempita automaticamente dall'**Object Wizard**. Ogni

elemento conterrà un **CLSID** e un nome di una classe C++. L'oggetto *_Module* legge la mappa e crea gli oggetti in base al loro **CLSID**.
Il file prosegue con la funzione **DllMain**. Essa semplicemente chiama i metodi **Init** e **Term** dell'oggetto *_Module*:

```
extern "C" BOOL WINAPI DllMain(HINSTANCE hInstance,
     DWORD dwReason, LPVOID /* lpReserved */)
{
     if (dwReason == DLL_PROCESS_ATTACH)
     {
          _Module.Init(ObjectMap, hInstance,
               &LIBID_MYOBJECTMODLib);
          DisableThreadLibraryCalls(hInstance);
     }
     else if (dwReason == DLL_PROCESS_DETACH)
          _Module.Term();
     return TRUE;  // tutto ok
}
```

Notiamo che **DllMain** chiama anche la funzione **DisableThreadLibraryCalls** per fare in modo che la funzione non sia chiamata ogni volta che un thread attacca la DLL.

Successivamente troviamo le quattro funzioni che una DLL COM deve avere:

```
STDAPI DllCanUnloadNow(void)
{
     return (_Module.GetLockCount() == 0) ? S_OK : S_FALSE;
}

STDAPI
DllGetClassObject(REFCLSID rclsid, REFIID riid, LPVOID* ppv)
{
     return _Module.GetClassObject(rclsid, riid, ppv);
}

STDAPI DllRegisterServer(void)
{
     // registra gli oggetti, la Type Library e
     // tutte le interfacce presenti in essa
     return _Module.RegisterServer(TRUE);
}

STDAPI DllUnregisterServer(void)
{
     return _Module.UnregisterServer(TRUE);
}
```

Queste funzioni non fanno altro che chiamare le corrispondenti funzioni dell'oggetto *_Module* che quindi sa come registrare, deregistrare, restituire la Class Factory e quando scaricare il modulo.

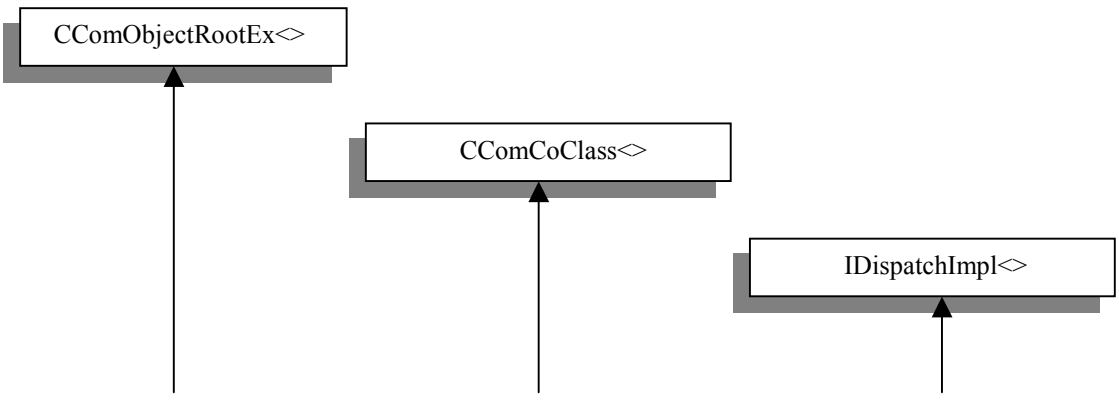
Dopo che abbiamo creato il modulo e capito più o meno come funziona, possiamo aggiungere gli oggetti. Il modo più facile per farlo è usare l'**ATL Object Wizard**. Basta selezionare la voce **New ATL Object…** dal menù **Insert** e poi selezionare la categoria **Object** e l'oggetto **Simple Object**. Premendo il tasto **Next >** si apre una finestra dove è possibile selezionare il nome del nuovo oggetto da aggiungere. Nella casella **Short Name** scriviamo *MyObject* e il **Wizard** riempirà il resto. Nella casella **Interface** cambiamo il nome scritto dal **Wizard** e scriviamo **IFoo** (la nostra prima interfaccia). Selezionando la cartella **Attributes** apparirà una finestra dove è possibile selezionare le proprietà del componente. Si può scegliere il modello di **threading**, il tipo di interfaccia e se si vuole supportare l'**aggregazione**. Le scelte di default vanno più che bene. Scegliendo il modello di **threading Apartment**, l'oggetto può essere usato da più **thread**, ma COM assicura che solo il **thread** che ha creato l'oggetto chiamerà sempre i suoi metodi, cioè ogni istanza dell'oggetto è **single-threaded**. In questo modo si possono avere più oggetti creati da **thread** diversi, così se ci sono dei dati globali nei metodi, devono essere trattati in modo **thread-safe.** Nel modello di oggetti **Apartment, la Class Factory** deve essere **multithread-safe**, ma non lo deve essere l'oggetto stesso. Siccome sarà **ATL** ad implementare la **Class Factory** per noi, non dobbiamo preoccuparci di questo.
Usiamo un'interfaccia **dual** per l'oggetto. Ripetiamo ancora che con questo tipo di interfaccia l'oggetto può essere chiamato sia tramite *vtable*, sia tramite **IDispatch::Invoke**. Le interfacce COM standard (quelle con la *vtable*) sono molto più veloci, quindi, quando possiamo, è sempre meglio usarle.
Dell'**aggregazione** abbiamo già parlato in precedenza. Se scegliamo di supportarla **ATL** farà le modifiche necessarie al componente in modo automatico e trasparente per il programmatore.
Nella stessa finestra di dialogo ci sono anche tre opzioni che si possono selezionare: **Support ISupportErrorInfo** che serve per le informazioni sugli errori (usata molto in **Visual Basic**), **Support Connection Points** che è usata principalmente per gli eventi, e **Free Threaded Marshaler** che è usata per certi tipi di controlli **multithreaded**. Premendo il tasto **OK**, il componente sarà creato automaticamente con tutte le opzioni selezionate, senza dover scrivere nessuna riga di codice. Verranno aggiunti al progetto alcuni file e verranno modificati degli altri. Due di questi sono i file che conterranno l'*header* e l'implementazione del componente e cioè **MyObject.h** e **MyObject.cpp** rispettivamente. Il secondo è quasi vuoto perché non abbiamo ancora implementato né le proprietà e né i metodi dell'oggetto. Il primo contiene invece la dichiarazione della classe che implementerà il componente:

```
class ATL_NO_VTABLE CMyObject :
     public CComObjectRootEx<CComSingleThreadModel>,
     public CComCoClass<CMyObject, &CLSID_MyObject>,
     public IDispatchImpl<IFoo,
          &IID_IFoo, &LIBID_MYOBJECTMODLib>
{
public:
     CMyObject()
     {
```

```
    }

DECLARE_REGISTRY_RESOURCEID(IDR_MYOBJECT)

DECLARE_PROTECT_FINAL_CONSTRUCT()

BEGIN_COM_MAP(CMyObject)
    COM_INTERFACE_ENTRY(IFoo)
    COM_INTERFACE_ENTRY(IDispatch)
END_COM_MAP()

// IFoo
public:

};
```

La classe è derivata da alcune *istanziazioni* di **template**. **CComObjectRootEx** implementa il **reference counting** dell'oggetto. **CComCoClass** si occupa della **Class Factory**. La derivazione più interessante è l'ultima, cioè **IDispatchImpl** che fornisce un'implementazione dell'interfaccia **duale** basata su **IFoo** che ha **IID_IFoo** come **ID** di interfaccia. Se avessimo avuto un'interfaccia normale (derivata da **IUnknown**), avremmo dovuto ereditare direttamente da **IFoo**.
La classe contiene anche alcune macro. **ATL_NO_VTABLE** ordina al compilatore di non generare una *vtable* per questa classe. Può essere usata solo per le classi che non verranno *istanziate*. Infatti non sarà la classe **CMyObject** ad essere *istanziata*, ma **ATL** deriverà un'altra classe da **CMyObject**, e sarà questa ad essere istanziata. Questa nuova classe sarà del tipo **CComObject<CMyObject>** di cui si può vedere una schema di derivazione nella Figura 2.18.

Il compito principale di **CComObject** è quello di fornire un'implementazione per i metodi di **IUnknown**. Queste *devono* essere fornite nella classe *più derivata* in modo che l'implementazione può essere condivisa con tutte le interfacce che derivano da **IUnknown** (nel nostro caso **IFoo** e **IGoo**). Comunque i metodi di **CComObject** si limitano a chiamare le implementazioni fornite da **CComObjectRootEx**. Un errore comune quando si usa **ATL** è quello di chiamare *new* sulla classe di implementazione (**CMyObject** nel nostro caso). Questo non funziona per le ragioni viste prima; quindi per creare un componente (anche dal server) si deve usare la funzione **CoCreateInstance**.



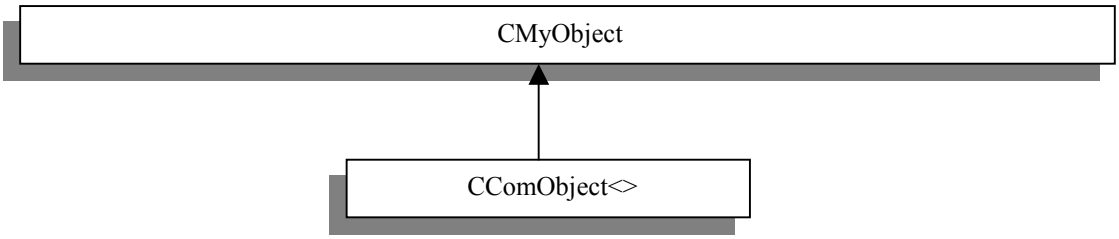


Figura 2.18 Schema di derivazione della classe **CComObject<CMyObject>.**

Un'altra cosa da notare nel file sono le macro che contengono le interfacce che il componente implementa (per il momento), e cioè **IFoo** e **IDispatch**. Non c'è nessuna voce per **IUnknown** perché è sottintesa. Le altre due macro servono rispettivamente per associare un file **.rgs** (**MyObject.rgs**) al componente che serve per la registrazione e la deregistrazione nel registro di Windows, e per cambiare il modo di costruire l'oggetto per essere sicuri che non venga cancellato accidentalmente.

L'ultimo file aggiunto dal **Wizard** è **MyObject.rgs** che contiene lo script per il codice di **ATL** per la manipolazione del registro. La maggior parte di esso corrisponde esattamente alle entrate del registro che devono essere aggiunte per permettere a COM di trovare il modulo che contiene il componente. Vediamone il contenuto:

```
HKCR
{
   MyObjectMod.MyObject.1 = s 'MyObject Class'
   {
      CLSID = s '{43B74E21-DB32-11d2-8B40-00400559C94F}'
   }
   MyObjectMod.MyObject = s 'MyObject Class'
   {
      CLSID = s '{43B74E21-DB32-11d2-8B40-00400559C94F}'
      CurVer = s 'MyObjectMod.MyObject.1'
   }
   NoRemove CLSID
   {
      ForceRemove {43B74E21-DB32-11d2-8B40-00400559C94F} =
         s 'MyObject Class'
      {
         ProgID = s 'MyObjectMod.MyObject.1'
         VersionIndependentProgID = s 'MyObjectMod.MyObject'
         ForceRemove 'ProgrammablÈ
         InprocServer32 = s '%MODULE%'
         {
            val ThreadingModel = s 'Apartment'
         }
         'TypeLib' =
            s '{98321601-DB33-11d2-8B40-00400559C94F}'
```

```
        }
    }
}
```

Questo script è usato sia per la registrazione che per la deregistrazione del componente. Per default quando si registra tutte le chiavi sono aggiunte, qualsiasi cosa ci possa essere già nel registro. La parola chiave **ForceRemove** modifica questo comportamento così che la chiave alla quale **ForceRemove** è applicata è rimossa (incluse le sottochiavi) prima di essere aggiunta di nuovo.
Quando si deregistra, per default tutte le chiavi che compaiono nello script sono rimosse (comprese le sottochiavi). È di fondamentale importanza escludere da questo la chiave **CLSID** usando la parola chiave **NoRemove**, questo per evitare che quando si deregistra il componente l'intero albero dei **CLSID** venga rimosso. Quindi è di fondamentale importanza non modificare questo script (a meno di sapere bene cosa si sta facendo). Per fortuna questo file viene generato automaticamente dal **Wizard**.

Facciamo una panoramica delle sottochiavi **HKCR**. All'inizio si trovano le estensioni dei file che sono state registrate da programmi diversi. Dopo queste vediamo un certo numero di altri nomi, la maggioranza dei quali è definita **ProgID** (**programmatic identifier**). Alcuni dei nomi non sono **ProgID**, ma chiavi speciali simili alla chiave **CLSID**. Queste chiavi mappano un **GUID** ad altre informazioni, per esempio ad un nome di file. Queste chiavi speciali comprendono quelle elencate di seguito:

- **AppID** Le sottochiavi di questa chiave vengono utilizzate per mappare un **AppID** (**application ID**) a un nome di server remoto. Questa chiave viene utilizzata da DCOM.
- **Component Categories** Questo ramo del registro mappa i **CatID** (**component category ID**) a una particolare categoria di componenti.
- **Interface** Questa chiave viene utilizzata per mappare **IID** a informazioni specifiche di un'interfaccia. Queste informazioni servono principalmente per utilizzare le interfacce attraverso i confini di processo.
- **Licenses** Questa chiave memorizza le licenze che concedono il permesso di utilizzare componenti COM.
- **TypeLib** Questa chiave mappa un **LibID** al nome del file dove è memorizzata la **Type Library**.

Vediamo in maggiore dettaglio i **ProgID**. La maggior parte delle sottochiavi che si trovano nel ramo **HKCR** del registro sono proprio **ProgID**. Questi mappano per il programmatore una stringa *friendly* a un **CLSID**. Alcuni linguaggi di programmazione, come **Visual Basic**, identificano i componenti in base ai **ProgID** e non in base ai **CLSID**. Non è garantita l'unicità dei **ProgID**, quindi un potenziale rischio è costituito dal conflitto dei nomi. D'altro canto è più facile lavorare con essi. Per convenzione i **ProgID** hanno il seguente formato:

```
<Programma>.<Componente>.<Versione>
```

Ecco alcuni esempi:

```
Visio.Application.3
Visio.Drawing.4
MyObjectMod.MyObject.1
```

L'ultimo esempio dovrebbe rendere più chiaro lo script visto prima. Questo formato però è solo una convenzione e quindi ci sono molti componenti che non la rispettano.
In molti casi al client non interessa a quale versione del componente si collega. Pertanto un componente di solito ha un **ProgID** che è indipendente dalla versione. La convenzione sui nomi per i **ProgID** indipendenti dalla versione consiste nell'omettere il numero finale. Si ha così questo formato:

```
<Programma>.<Componente>
```

I **ProgID** e quelli indipendenti dalla versione, vengono elencati come sottochiavi del **CLSID** del componente. Tuttavia lo scopo principale di un **ProgID** è quello di ottenere il corrispondente **CLSID**. Cercare un **ProgID** sotto la voce di ogni **CLSID** non è efficiente. Pertanto il **ProgID** viene anche elencato direttamente nel ramo **HKCR** e come sottochiave viene aggiunto il **CLSID**. Il **ProgID** indipendente dalla versione ha anche una sottochiave aggiuntiva, **CurVer**, che contiene il **ProgID** della versione corrente del componente. Una volta inserite queste informazioni nel registro, ottenere un **CLSID** da un **ProgID**, o viceversa, è molto facile. La libreria COM fornisce due funzioni per tale scopo: **CLSIDFromProgID** e **ProgIDFromCLSID**.

Vediamo adesso i cambiamenti fatti dal **Wizard** ai file già esistenti. Al file **MyObjectMod.cpp** è stata aggiunta questa riga:

```
#include "MyObject.h"
```

Inoltre è stata aggiunta una voce alla **Object Map** che ora è la seguente:

```
BEGIN_OBJECT_MAP(ObjectMap)
     OBJECT_ENTRY(CLSID_MyObject, CMyObject)
END_OBJECT_MAP()
```

Il file **.idl** ha adesso una voce per l'interfaccia **IFoo**, che è derivata da **IDispatch**, e una voce **coclass** nella **Type Library** per la classe **CMyObject**. Il file **.rc** ha alcuni piccoli cambiamenti e una linea che include lo script per il registro tra le risorse della DLL.

Aggiungiamo adesso le altre due interfacce **IFoo2** (che deriva da **IFoo**) e **IGoo** (che deriva direttamente da **IUnknown**). Siccome non c'è modo per un client che usa il **late-binding** (la **dispinterface**) di cambiare interfaccia (può usare solamente quella di default) non è necessario derivare anche **IGoo** da **IDispatch**.
Sfortunatamente, non esiste nessun **Wizard** che aggiunge delle ulteriori interfacce ad un oggetto già esistente, quindi si deve fare tutto il lavoro a mano. Dobbiamo

quindi aggiungere le due interfacce nel file **MyObjectMod.idl**, così alla fine sarà il seguente:

```
[
     object,
     uuid(4ECECC21-D25E-11d2-8B40-00400559C94F),
     helpstring("IFoo interface"),
     pointer_default(unique)
]
interface IFoo : IDispatch
{
     [id(1), helpstring("method Func1")] HRESULT Func1();
     [id(2), helpstring("method Func2")]
                    HRESULT Func2([in] int nCount);
};

[
     object,
     uuid(4ECECC22-D25E-11d2-8B40-00400559C94F),
     helpstring("IFoo2 interface"),
     pointer_default(unique)
]
interface IFoo2 : IFoo
{
     [id(3), helpstring("method Func3")]
                    HRESULT Func3([out, retval] int *pout);
};

[
     object,
     uuid(4ECECC23-D25E-11d2-8B40-00400559C94F),
     helpstring("IGoo interface"),
     pointer_default(unique)
]
interface IGoo : IUnknown
{
     [helpstring("method Gunc")] HRESULT Gunc();
};
```

Le cose da notare sono la derivazione di **IFoo** da **IDispatch** e di **IFoo2** da **IFoo**, e i **DISPID** che precedono ogni metodo (*[id(1)]*, ecc.). Il **DISPID** di **Func3** è *id(3)* perché **IFoo2** deriva da **IFoo**, quindi *id(1)* e *id(2)* sono già assegnati. È molto importante mettere un **DISPID** diverso per ogni funzione della stessa interfaccia (considerando anche l'ereditarietà) perché altrimenti si ottiene un errore in fase di compilazione.
Devo inoltre modificare la dichiarazione di **coclass** per indicare che l'oggetto implementa le nuove interfacce. La modifica è la seguente:

```
coclass MyObject
{
     interface IFoo;
```

```
      [default] interface IFoo2;
      interface IGoo;
}
```

Ho scelto **IFoo2** come interfaccia di default così un client **late-bound** può usare le funzioni aggiuntive.

Nel file CMyObject.h dobbiamo cambiare la dichiarazione della classe. Prima avevamo:

```
class ATL_NO_VTABLE CMyObject :
   public CComObjectRootEx<CComSingleThreadModel>,
   public CComCoClass<CMyObject, &CLSID_MyObject>,
   public IDispatchImpl<IFoo,
      &IID_IFoo, &LIBID_MYOBJECTMODLib>
```

dopo la modifica abbiamo:

```
class ATL_NO_VTABLE CMyObject :
   public CComObjectRootEx<CComSingleThreadModel>,
   public CComCoClass<CMyObject, &CLSID_MyObject>,
   public IDispatchImpl<IFoo2,
      &IID_IFoo2, &LIBID_MYOBJECTMODLib>,
   public IGoo
```

Siccome **IFoo2** è derivata da **IFoo**, abbiamo **IFoo2** e non **IFoo** nella lista dell'ereditarietà (**IFoo** è nella lista implicitamente), e poiché **IFoo2** è un'interfaccia duale, usiamo il template **IDispatchImpl**, invece di ereditare direttamente da essa.
Dobbiamo anche cambiare la mappa delle interfacce da:

```
BEGIN_COM_MAP(CMyObject)
      COM_INTERFACE_ENTRY(IFoo)
      COM_INTERFACE_ENTRY(IDispatch)
END_COM_MAP
```

in:

```
BEGIN_COM_MAP(CMyObject)
      COM_INTERFACE_ENTRY(IFoo)
      COM_INTERFACE_ENTRY(IFoo2)
      COM_INTERFACE_ENTRY(IGoo)
      COM_INTERFACE_ENTRY(IDispatch)
END_COM_MAP
```

Un metodo si aggiungere ad un'interfaccia sia come abbiamo fatto prima, cioè modificando il file **.idl**, sia *cliccando* con il pulsante destro del mouse sull'icona dell'interfaccia nella finestra **Class View**, e selezionando la voce del menù contestuale **Add Method…** . Si accede così ad una finestra di dialogo dove è possibile selezionare il tipo di ritorno del metodo, il suo nome e i suoi parametri.

Con la stessa operazione, ma selezionando la voce **Add Property…** dal menù contestuale si può aggiungere una proprietà all'interfaccia.
Le interfacce COM non possono avere variabili al loro interno, così vengono simulate con le proprietà. Una proprietà è una coppia di metodi marcati con gli attributi **IDL** *[propget]* e *[propput]* che servono rispettivamente per leggere e settare il valore della proprietà (è implementata nella classe con una variabile e con i due metodi corrispondenti per leggerla e per settarla). Facciamo un esempio. Se volessi aggiungere all'interfaccia **IGoo** la proprietà **StatoInterno** di tipo *long*, dovrei agire in questo modo. Dopo aver scelto la voce **Add Property…** dal menù contestuale, si apre una finestra dove si può scegliere il tipo della proprietà (scriviamo *long*), il nome (scriviamo *StatoInterno*), e le funzioni che si vogliono su questa proprietà (selezioniamo sia **Get Function** che **Set Function**). Dopo aver cliccato sul pulsante **OK** al file **.idl** viene aggiunto del codice e l'interfaccia IGoo avrà il seguente aspetto:

```
[
     object,
     uuid(4ECECC23-D25E-11d2-8B40-00400559C94F),
     helpstring("IGoo interface"),
     pointer_default(unique)
]
interface IGoo : IUnknown
{
     [helpstring("method Gunc")] HRESULT Gunc();
     [propget, helpstring("property StatoInterno")]
          HRESULT StatoInterno([out] retval] long *pVal);
     [propput, helpstring("property StatoInterno")]
          HRESULT StatoInterno([in] long newVal);
};
```

L'implementazione dei metodi delle, **Func1**, **Func2**, **Func3** e **Gunc** sono identiche a quelle già viste in precedenza, e cioè:

```
// IFoo
STDMETHODIMP CMyObject::Func1()
{
     m_iInternalValue++;
     if (m_iInternalValue % 3 == 0)
          MessageBeep((UINT) -1);
     return S_OK;
}

STDMETHODIMP CMyObject::Func2(int nCount)
{
     m_iInternalValue = nCount;
     return S_OK;
}

// IFoo2
STDMETHODIMP CMyObject::Func3(int *pout)
```

```
{
     *pout = m_iInternalValue;
     return S_OK;
}

// IGoo
STDMETHODIMP CMyObject::Gunc(void)
{
     MessageBeep((UINT) -1);
     return S_OK;
}
```

Inseriamo queste implementazioni nel file **CMyObject.cpp** e quindi possiamo compilare il tutto scegliendo la voce **Build MyObjectMod.dll** dal menu **Buid**. Otterremo così non solo il componente, ma anche la sua registrazione dato che nel *makefile* viene aggiunto automaticamente un comando che si occupa appunto di registrare il componente.

2.18 Il Client

Fino a questo punto ci siamo occupati di come realizzare il componente e di come inserirlo in un server, adesso vedremo come realizzare un client per il nostro componente. Faremo degli esempi in **C++**, **Visual C++**, **Visual Basic** e **Visual J++**.

2.18.1 Un client C++

Realizzare un client in C++ non è molto difficile. Come abbiamo visto in precedenza il compilatore **MIDL** produce un file *header* che ha come base del nome lo stesso nome del file **.idl** (nel nostro caso da **MyObject.idl** otteniamo **MyObject.h**, nel caso del server **Visual C++** da **MyObjectMod.idl** otteniamo **MyObjectMod.h**). Abbiamo usato questo file per costruire il componente, ma ci serve anche per costruire il client. Il **MIDL** ha anche generato un file per la definizione dei **GUID**. Questo file ha la base del nome come quella del file **.idl** e un'estensione **_i.c** (nel nostro esempio si chiama **MyObject_i.c**).
Quello che dobbiamo fare è includere l'*header* nel client come segue:

```
#include "MyObject.h"
```

Invece di includere anche il file **MyObject_i.c**, lo inseriamo direttamente nel progetto, cioè lo aggiungiamo al *makefile*, in modo che venga compilato e linkato con il resto del client.

Prima di usare qualsiasi oggetto COM, dobbiamo inizializzare la libreria COM chiamando la funzione **CoInizialize** e, prima che il programma termini, dobbiamo

chiamare **CoUninitialize**. L'agomento *NULL* è richiesto, e dobbiamo testare il valore di ritorno di **CoInitialize** nel seguente modo:

```
HRESULT hr = CoInitialize(NULL);
if (FAILED(hr))
{
     cout << "CoInitialize fallita" << hr << endl;
     exit(1);
}
else
     cout << "CoInizialize OK" << endl;
}
```

Tralasciando i canoni della programmazione strutturata, possiamo aggiungere una *label* alla fine del programma a cui saltare se la creazione del componente fallisce, o quando il client deve terminare. Nulla vieta di usare una serie di *if* annidati. Noi opteremo per la prima soluzione:

```
// . . . resto del programma

Uninit:
     CoUninitialize();

} // end del main()
```

Se la nostra applicazione è **multithreaded** e vuole usare degli oggetti che usano il modello di **thread free**, dobbiamo usare la funzione **CoInizializeEx** al posto di **CoInizialize**. Non è comunque il nostro caso.
Non dobbiamo esplicitamente includere gli *header* di COM perché il file *header* generato dal **MIDL** (che abbiamo incluso nel client) li include automaticamente.

Una volta che la libreria COM è stata inizializzata, possiamo creare l'oggetto:

```
IFoo* pFoo;
hr = CoCreateInstance(CLSID_MyObject, NULL, CLSCTX_ALL,
          IID_IFoo, reinterpret_cast<void **>(&pFoo));
if (FAILED(hr))
{
     cout << "CoCreateInstance fallita" << hr << endl;
     goto Uninit;
}
else
     cout << "CoCreateInstance OK" << endl;
```

Gli header generati dal **MIDL** si occupano di tutte le dichiarazioni: il tipo puntatore all'interfaccia **IFoo**, il **GUID** per l'**ID** della classe e quello per l'**ID** dell'interfaccia, ecc. È sicuramente una notevole semplificazione. Dei parametri di **CoCreateInstance** abbiamo già parlato in precedenza.
In COM non avremo mai un puntatore all'oggetto; avremo invece un puntatore ad una delle sue interfacce. Per questo specifichiamo che vogliamo l'interfaccia **IFoo**.

L'ultimo parametro è un puntatore dove memorizzeremo il puntatore all'interfaccia restituito da **CoCreateInstance**. Se la creazione fallisce (sia perché l'oggetto non può essere creato, sia perché l'interfaccia richiesta non è supportata), viene restituito nel puntatore il valore *NULL*. Il nuovo operatore *reinterpret_cast* è la maniera migliore per fare il *cast* del puntatore al valore *void***, infatti esso indica che la nostra intenzione è quella di vedere i bit del puntatore in modo diverso (tipo diverso) e non vogliamo cambiarne il valore in ogni modo.
Se la creazione fallisce chiudiamo la libreria COM con un salto a *Uninit*.

Chiamare i metodi dell'interfaccia è banale. Basta fare la chiamata tramite il puntatore all'interfaccia:

```
pIFoo->Func1();
pIFoo->Func2(8);
```

Anche qui avremmo dovuto testare il valore di ritorno *HRESULT*, ma siccome usiamo un server **InProc** non è una cosa cruciale (diverso è il caso dei server **OutOfProc** o **Remote** che possono più facilmente causare malfunzionamenti). Per esempio in un server **Remote** ci potrebbe essere un errore di rete, oppure il server **OutOfProc** potrebbe terminare accidentalmente. Quindi è sempre una buona regola fare le chiamate in questo modo:

```
hr = pIFoo->Func1();
if (FAILED(hr)) // procedura di ricovery per Func1 //;

hr = pIFoo->Func2(8);
if (FAILED(hr)) // procedura di ricovery per Func2 //;
```

Il componente espone le sue funzionalità attraverso tre interfacce (quattro se consideriamo anche **IUnknown**). Per questo per usare completamente il componente, dobbiamo passare anche alle altre interfacce. Questa operazione viene svolta dalla funzione **QueryInterface**:

```
IGoo* pIGoo;
hr = pIGoo->QueryInterface(IID_IGoo,
                reinterpret_cast<void **>(&pIGoo));

if (FAILED(hr))
{
     cout << "QueryInterface fallita" << hr << endl;
     goto ReleaseIFoo;
}
else
     cout << "QueryInterface OK" << endl;
```

Passiamo a **QueryInterface** l'**ID** della nuova interfaccia e un puntatore dove memorizzare il puntatore all'interfaccia restituito dalla funzione. Se la chiamata fallisce dobbiamo rilasciare il puntatore **IFoo** con il salto a *ReleaseIFoo* (che vedremo in seguito). Possiamo adesso chiamare i metodi dell'interfaccia **IGoo**:

```
hr = pIGoo->Gunc();
if (FAILED(hr)) // procedura di ricovery per Gunc //;
```

Se avessimo voluto usare anche **IFoo2** avremmo dovuto lo stesso fare una chiamata a **QueryInterface**, infatti anche se **IFoo2** deriva da **IFoo**, COM le vede come due interfacce completamente separate.

Dopo che abbiamo usato l'oggetto (o gli oggetti), dobbiamo rilasciare i puntatori per permettere a COM di scaricare la DLL o di terminare l'EXE (se necessario). Questo si fa con del codice simile al seguente:

```
cout << "Tutto OK" << endl;
pIGoo->Release();

ReleaseIFoo:
     pIFoo->Release();

// fine del programma
Uninit:
     CoUninitialize();

} // end del main()
```

Abbiamo separato le due chiamate a **Release** con una *label* perché se la chiamata a **QueryInterface** per **IGoo** fosse fallita, avremmo dovuto rilasciare solo il puntatore a **IFoo** e quindi non anche quello a **IGoo**.

2.18.2 Un client Visual C++

Usando il **Visual C++ 5.0** o successivo, si possono usare una serie di facilitazioni che rendono l'uso di oggetti COM facile come in Visual Basic o Visual J++. Prima fra tutte gli **smart pointer** (puntatori intelligenti che gestiscono in modo trasparente tutte le funzioni di **IUnknown**). Questi **smart pointer** sono creati quando si usa la nuova direttiva *import* per importare una **Type Library**. Le **Type Library** che si possono importare sono le stesse che si possono caricare con la funzione **LoadTypeLib**. Di solito sono file che hanno l'estensione **.tlb**, **.odl**, **.EXE**, **.DLL**, **.OCX**, ecc.
Tutto quello che si deve fare è importare la **Type Library** e usare gli **smart pointer** che sono stati creati per noi.

Con la direttiva *import* il **Visual C++** creerà due file che includerà automaticamente nel progetto. Il nome di questi due file avrà la stessa base del nome della **Type Library**, ma avranno le estensioni **.tlh** e **.tli**. Nel nostro caso con la riga:

```
#import "MyObjectMod.tlb"
```

otterremo i file **MyObjectMod.tlh** e **MyObjectMod.tli** che verranno memorizzati nella directory di output del progetto (di solito la directory **Debug**). Vediamo cosa contengono.
Il file **.tlh** contiene le seguenti dichiarazioni:

- Dichiarazioni di strutture con *declspec(uuid("<GUID>"))* che servono per associare i **GUID** alle classi e alle interfacce. Questo ci permette di usare l'operatore **uuidof** per ottenere il **GUID** di una classe o di un'interfaccia, con una istruzione del genere *__uuidof(MyObject)*.
- La definizione degli **smart pointer** per ogni puntatore a interfaccia usando la macro **_COM_SMARTPTR_TYPEDEF**. Questi **smart pointer** gestiscono le chiamate a **AddRef**, **Release** e **QueryInterface** in modo automatico e trasparente. Inoltre permettono di creare oggetti senza chiamare esplicitamente **CoCreateInstance**. La dichiarazione dello **smart pointer** per l'interfaccia **IFoo** è:

  ```
  _COM_SMARTPTR_TYPEDEF(IFoo, __uuidof(IFoo));
  ```

  che il compilatore espande in:

  ```
  typedef _com_ptr_t<_com_IIID<IFoo, __uuidof(IFoo)> >
     IFooPtr
  ```

  Questo comando creerà la classe **smart pointer** *IFooPtr* che si occuperà di **CoCreateInstance** e di tutte le funzioni di **IUnknown**.
- La definizione di classi **wrapper** che si occuperanno della gestione degli errori, dei metodi e delle proprietà delle interfacce. Non abbiamo proprietà nel nostro esempio, ma se le avessimo avute, potremmo usarle allo stesso modo di **Visual Basic**. Facciamo un esempio. Supponiamo di avere una proprietà che si chiama *Stato*. Il **Visual C++** crea automaticamente lo scheletro di due funzioni che la gestiscono, *get_Stato* e *set_Stato* (questo nell'implementazione dell'interfaccia). Con la classe **wrapper** creata possiamo usare direttamente la sintassi:

  ```
  pIFoo->Stato = 5;
  int val = pIFoo->Stato;
  ```

  invece delle tradizionali:

  ```
  pIFoo->set_Stato(5);
  int val;
  pIFoo->get_Stato(&val);
  ```

  Inoltre gli eventuali errori contenuti nel valore di ritorno **HRESULT** vengono mappati in opportune eccezioni di tipo **com_error**. Così la dichiarazione di **IFoo** diventa:

  ```
  struct
  __declspec(uuid("4ECECC21-D25E-11d2-8B40-00400559C94F"))
  ```

```
IFoo : IUnknown
{

   // metodi wrapper per la gestione degli errori
   HRESULT Func1();
   HRESULT Func2(int nCount);

   // metodi raw forniti dall'interfaccia
   virtual HRESULT __stdcall raw_Func1() = 0;
   virtual HRESULT __stdcall raw_Func2(int nCount) = 0;

};
```

Come vediamo vengono cambiati i nomi reali delle funzioni dell'interfaccia, facendoli precedere dal prefisso *raw_*. Sono i metodi **raw** che mappano le funzioni dell'interfaccia. Infatti i metodi **wrapper** chiamano quelli **raw**.

Tutte queste dichiarazioni sono contenute in un **namespace** che ha lo stesso nome della **LIBRARY** contenuta nella **Type Library**, quindi tutti i nomi si devono qualificare in questo modo:

```
MYOBJECTMODLib::IFooPtr *pIFoo;
```

oppure usando **namespace**:

```
using namespace MYOBJECTMODLib

IFooPtr *pIFoo;
```

Si possono passare dei parametri alla direttiva *import* per guidare il processo di creazione delle dichiarazioni precedenti. Quelle che abbiamo visto sono quelle generate per default.

Il file **.tli** contiene semplicemente l'implementazione di tutte le funzioni **wrapper** contenute nel file **.tlh**. Per esempio per l'interfaccia **IFoo** contiene le funzioni **wrapper** che chiamano le corrispondenti funzioni **raw** e sollevano un'eccezione se **HRESULT** contiene un errore:

```
inline HRESULT IFoo::Func1()
{
     HRESULT _hr = raw_Func1();
     if (FAILED(_hr))
          _com_issue_errorex(_hr, this, __uuidof(this));
     return _hr;
}

inline HRESULT IFoo::Func2(int nCount)
{
     HRESULT _hr = raw_Func2(nCount);
```

```
    if (FAILED(_hr))
        _com_issue_errorex(_hr, this, __uuidof(this));
    return _hr;
}
```

Vediamo adesso come creare il client. Per prima cosa dobbiamo usare la direttiva **import** per far accedere il compilatore alla **Type Library** come abbiamo visto prima. Successivamente dobbiamo inizializzare la libreria COM con la funzione **CoInitialize** o **CoInitializeEx** (si possono usare anche le corrispondenti funzioni **OLE AfxOleInit**).

Creare un oggetto è molto semplice, si deve solamente creare uno **smart pointer** passandogli il **GUID** dell'oggetto che vogliamo creare. Si usa in questo caso un *overload* del costruttore che si occupa di chiamare **CoCreateInstance**:

```
IFooPtr pIFoo(__uuidof(MyObject));
```

In questo caso abbiamo supposto di usare *using namespace. . .* in modo da poter usare i nomi direttamente.

Dopo che l'oggetto è stato creato, possiamo chiamare i suoi metodi. La sintassi rimane invariata:

```
pIFoo->Func1();
pIFoo->Func2(5);
```

Se abbiamo delle proprietà nell'interfaccia, le possiamo usare direttamente:

```
pIFoo->Stato = 5;  // chiama il metodo set
int val = pIFoo->Stato;  // chiama il metodo get
```

Il compilatore automaticamente cambia questi riferimenti con le appropriate funzioni **set_** e **get_**.

Come abbiamo visto qualsiasi chiamata all'oggetto può fallire. Per default un'eccezione **com_error** è sollevata ogni volta che si verifica un errore. Possiamo usare un blocco *try/catch* per catturare le eccezioni:

```
try {
    IFooPtr pIFoo(__uuidof(MyObject));
    pIFoo->Func1();
}
catch (_com_error e)
{
    AfxMessageBox(e.ErrorMessage());
}
```

L'uso delle eccezioni rende il codice più pulito e facile da capire, ma si paga un dazio dalla parte delle dimensioni del codice e della sua efficienza.

Usare una differente interfaccia è molto semplice: basta creare un nuovo **smart pointer** del tipo della nuova interfaccia e inizializzarlo con il puntatore alla vecchia. Il costruttore o l'operatore di assegnamento chiamerà automaticamente **QueryInterface**. Per esempio possiamo scrivere:

```
IGooPtr pIGoo(__uuidof(MyObject));  // crea l'oggetto
pIGoo->Gunc();

IFooPtr pIFoo = pIGoo; // chiama QueryInterface
                       // sullo stesso oggetto
pIFoo->Func1();
```

In questo caso il costruttore di *IFooPtr* che prende uno **smart pointer** come parametro, chiama **QueryInterface**. Se fallisce viene sollevata un'eccezione.

Alla fine del programma dobbiamo aggiungere la solita chiamata a **CoUninitialize** per chiudere la libreria COM. Se abbiamo scelto di usare la **MFC** nel client, questa chiamata non è necessaria perché è la **MFC** a farla automaticamente.

2.18.3 Un client Visual Basic

Il client **Visual Basic** è sicuramente quello più semplice. Per prima cosa dobbiamo aggiungere un *reference* al nostro progetto. Selezionando la voce **Reference…** dal menù **Project**, si apre una finestra dove dobbiamo selezionare nella **list box** la voce **MyObjectMod 1.0 Type Lybrary** e quindi premiamo il tasto **OK**.

Disegnamo una **Form** come mostra la Figura 2.19. Adesso possiamo scrivere il codice necessario per creare e gestire il componente. Per prima cosa, nella sezione delle dichiarazioni, dichiariamo un *reference* all'oggetto:

```
Private obj As MyObject
```

Creiamo poi l'oggetto nel metodo **Load** della **Form**:

```
Private Sub Form1_Load()
     Set obj = New MyObject
End Sub
```

Con queste istruzioni diciamo a **Visual Basic** di chiamare **CoCreateInstance** per creare l'oggetto (viene restituita l'interfaccia che abbiamo indicato nel server come quella di default).

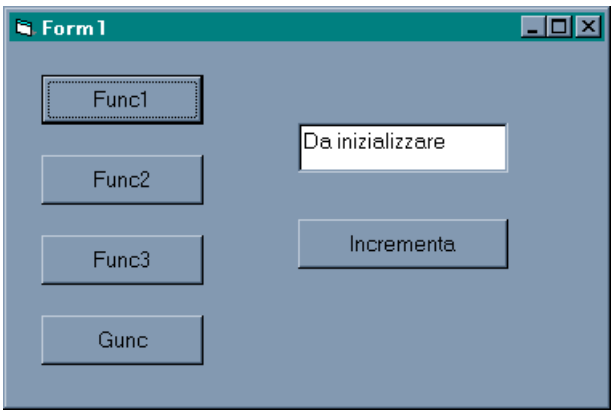


Figura 2.19 Form per il test del client in **Visual Basic**.

Adesso che l'oggetto è stato creato, aggiungiamo i gestori dei pulsanti che si occuperanno di chiamare i metodi dell'interfaccia di default (che è **IFoo**):

```
Private Sub Func1_Click()
     obj.Func1
End Sub

Private Sub Func2_Click()
     If IsNumeric(Text1) Then obj.Func2(Text1)
End Sub
```

Per chiamare i metodi nelle altre interfacce, dobbiamo poter accedere alle altre interfacce. Questo si può fare creando una variabile del tipo corretto per l'interfaccia e poi settarla al puntatore all'oggetto (facendo così in modo che **Visual Basic** chiami **QueryInterface** sull'oggetto). Il codice è questo:

```
Dim Foo2 As IFoo2
Set Foo2 = obj  ' cambia interfaccia con QueryInterface
```

Possiamo quindi chiamare i metodi di **IFoo2** usando la variabile **Foo2**. Sapendo questo il codice seguente diventa facile da capire:

```
Private Sub Func3_Click()
     Dim Foo2 As IFoo2
     Set Foo2 = obj
     Text1 = Foo2.Func3
End Sub

Private Sub Gunc_Click()
     Dim Goo As IGoo
     Set Goo = obj
     Goo.Gunc
End Sub
```

Alla fine aggiungiamo il gestore del tasto **Incrementa**:

```
Private Sub Incrementa_Click()
     If IsNumeric(Text1) Then Text1 = Text1 + 1
End Sub
```

Con il **late binding** si può accedere lo stesso a questo oggetto, ma si possono usare solo i metodi dell'interfaccia di default (**IFoo**). Il **late binding** lavora male con le interfacce multiple, ed è molto più lento. Per usare il **late bindind** avrei dovuto usare del codice simile a questo:

```
Private obj As Object

Private Sub Form1_Load()
     Set obj = New MyObject
End Sub
```

Non possiamo però poi cambiare interfaccia con:

```
Dim Goo As IGoo
Set Goo = obj
```

2.18.4 Un client Visual J++

Vediamo come realizzare un client con **Visual J++ 6.0**. Per prima cosa dobbiamo creare un nuovo progetto di tipo **Windows EXE** (è una nuova opzione aggiunta con questa versione). Aggiungiamo una classe che rappresenterà l'oggetto COM selezionando la voce **Add COM Wrapper** dal menù **Project**. Importiamo la classe con:

```
import MyObjectMod.*
```

Una volta fatto questo possiamo creare l'oggetto:

```
IFoo myObj = new MyObject();
```

È da notare che creiamo un oggetto **MyObject**, ma lo assegnamo al tipo **IFoo**, questo perché **Visual J++** non supporta il concetto d'interfaccia di default.
A questo punto possiamo chiamare i metodi di **IFoo**:

```
myObj.Func2(5);
myObj.Func1();
```

Se vogliamo usare i metodi di **IFoo2** e di **IGoo**, dobbiamo creare delle altre variabili del tipo giusto e assegnare l'oggetto ad esse:

```
IFoo2 myFoo2 = (IFoo2)myObj;
```

```
int a = myFoo2.Func3();

IGoo myGoo = (IGoo)myFoo2;
myGoo.Gunc();
```

Come possiamo vedere, usare gli oggetti COM da **Visual J++** è facile come quando li usiamo da **Visual Basic**.

Capitolo 3

CORBA

In questo capitolo daremo una descrizione delle caratteristiche principali dell'architettura CORBA e descriveremo l'uso di **Mico** (una sua implementazione). Dato che la specifica CORBA ha lasciato in alcuni punti delle lacune sull'implementazione e dato che i vari produttori hanno dato delle soluzioni proprietarie, descriveremo l'architettura CORBA in linea generale. I dettagli del codice saranno trattati di tanto in tanto facendo degli esempi in C++ usando l'implementazione fornita da Mico.

3.1 Common Object Request Broker Architecture

La **Common Object Request Broker Architecture** o CORBA è uno standard industriale che definisce una specifica ad alto livello per i sistemi distribuiti. È stata proposta dall'**OMG** (**Object Menagement Group**), un consorzio industriale il cui scopo è quello di creare un'infrastruttura ad oggetti veramente indipendente dalla piattaforma. Di questo consorzio fanno parte venditori di sistemi informatici, università e istituzioni. Fra i soci illustri ci sono: la Sun, l'IBM, la DEC, l'HP, ecc. Un'assenza di rilievo è quella della Microsoft, che ha sviluppato il sistema concorrente COM.

CORBA automatizza molte problematiche della programmazione in rete, come ad esempio la registrazione degli oggetti, la localizzazione, l'attivazione, la gestione degli errori, il **marshalling** e **demarshalling** dei parametri e l'invocazione delle operazioni.

La caratteristica principale di CORBA è che è un'architettura basata sugli oggetti. Gli oggetti CORBA presentano molte funzionalità e caratteristiche degli altri sistemi ad oggetti, incluse l'ereditarietà (sia delle interfacce che dell'implementazione) e il polimorfismo. Ciò che rende CORBA veramente interessante è che tutte queste caratteristiche non vengono meno nemmeno quando si usano dei linguaggi di programmazione che non sono orientati agli oggetti, come ad esempio il C e il COBOL.

La Figura 3.1 mostra una visione generale dell'architettura CORBA.

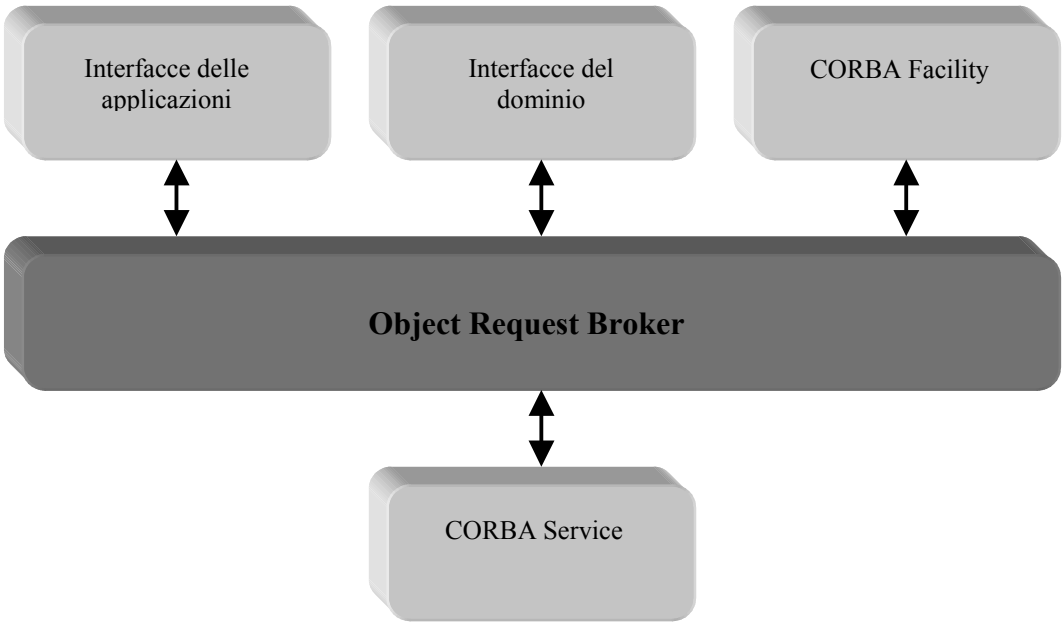


Figura 3.1 Architettura generale di CORBA.

Vediamo una piccola descrizione dei componenti:

- L'**Object Request Broker** o **ORB** è il cuore di CORBA e fornisce tutte le sue funzioni essenziali. Ne parleremo dettagliatamente nei paragrafi successivi.
- I **CORBA Service** sono delle interfacce, indipendenti dal dominio e dall'ORB, che forniscono dei servizi di uso generale. Un esempio di servizio è il meccanismo che permette ai client di cercare i server di cui hanno bisogno. Per questo scopo CORBA fornisce due **Service**: il **Naming Service** che permette di cercare gli oggetti in base al loro nome, e il **Trading Service** che invece permette di scoprire i server in base alle loro proprietà.
- Le **CORBA Facility** sono una collezione di interfacce che forniscono dei servizi di uso diretto per le applicazioni. Un esempio è il **Distributed Document Component Facility** o **DDCF**, basato sull'**OpenDoc**, che permette la presentazione e lo scambio di oggetti basati su un modello documento.
- Le **interfacce del dominio** sono simili ai Service e alle Facility, ma forniscono dei servizi che sono specifici di un dominio.
- Le **interfacce delle applicazioni** sono sviluppate appositamente per una specifica applicazione. Non fanno parte ovviamente della specifica CORBA, ma sono definite dai programmatori.

3.2 L'Object Request Broker

Il cuore di CORBA è l'**Object Request Broker** o **ORB** che è un componente software che serve a facilitare la comunicazione fra gli oggetti. Svolge il suo compito fornendo una serie di funzioni, la più importante delle quali è quella di localizzare gli oggetti remoti e restituire un riferimento ad essi. Un altro servizio fornito dall'ORB è il **marshaling** e il **demarshaling** dei parametri e dei valori di ritorno nell'invocazione dei metodi remoti.

Permette agli oggetti di fare le loro richieste e di ricevere le risposte trasparentemente, indipendentemente dalla loro locazione locale o remota. Il client non è a conoscenza dei meccanismi della comunicazione, dell'attivazione e della memorizzazione degli oggetti server. Nella specifica 1.1 di CORBA, introdotta nel 1991, veniva definito solo l'IDL, il mapping sui vari linguaggi e le API per l'interfacciamento con l'ORB. Rimanevano però le limitazioni sulla comunicazione tra gli ORB prodotti da diversi distributori. A partire dalla specifica 2.0 questi problemi sono stati eliminati.

Un ORB CORBA fornisce in pratica tutti quei meccanismi necessari alla comunicazione che non fanno parte né del client e né del server. Permette, inoltre, agli oggetti di individuarsi a vicenda a tempo di esecuzione e di invocare i loro servizi.

L'ORB è molto più sofisticato di tutte le altre forme di comunicazione client/server che sono attualmente presenti sul mercato. I suoi principali vantaggi sono i seguenti:

- **Invocazione dinamica e statica dei metodi**. L'ORB permette di definire le invocazioni dei metodi sia a tempo di compilazione che a tempo di esecuzione. Gli oggetti hanno la possibilità di fornire tutte le informazioni necessarie per la creazione di una richiesta. Si hanno così sia i vantaggi del controllo sui tipi a tempo di compilazione, sia la massima flessibilità associata al collegamento dinamico a tempo di esecuzione.
- **Mapping sui linguaggi ad alto livello**. L'ORB permette di invocare i metodi di un oggetto usando un qualsiasi linguaggio ad alto livello. Non si preoccupa di sapere con quale linguaggio è stato scritto il server. CORBA separa l'interfacce dall'implementazione e fornisce un meccanismo, l'IDL, per definire in modo quasi naturale le richieste che i client possono fare agli oggetti, scavalcando i limiti del linguaggio d'implementazione e del Sistema Operativo usato.
- **Sistema autodescrittivo**. CORBA fornisce dei *metadati* a tempo di esecuzione per descrivere ogni interfaccia conosciuta al sistema. Ogni ORB deve supportare un'**Interface Repository** che contiene, in tempo reale, le informazioni che descrivono le funzioni, e i loro parametri, che un server fornisce. Il client usa i *metadati* per scoprire come invocare un servizio a tempo di esecuzione. I *metadati* sono generati automaticamente sia dal compilatore IDL, sia da quegli strumenti che permettono di generare delle definizioni in IDL a partire da un linguaggio orientato agli oggetti (come ad esempio il compilatore MetaWare C++ che genera l'IDL a partire da delle classi C++).

- **Trasparenza sulla locazione**. Un ORB può funzionare da solo su un portatile, o può essere connesso a tutti gli altri ORB presenti nel mondo usando il servizio **Internet Inter ORB Protocol** introdotto a partire dalla versione 2.0 di CORBA. Un ORB può fare da mediatore tra gli oggetti di un singolo processo, di diversi processi che girano sulla stessa macchina e di diversi processi che girano su macchine diverse indipendentemente dal Sistema Operativo. Il tutto viene gestito in modo trasparente e a diversi livelli di astrazione. In generale un programmatore CORBA non si deve preoccupare della locazione del server, della sua attivazione, dell'ordine dei byte di un certo Sistema Operativo. È l'ORB che si occupa di tutto questo.
- **Polimorfismo**. Al contrario delle altre forme di comunicazione, un ORB non si limita a invocare una funzione remota, ma chiama una funzione di uno specifico oggetto. Questo significa che la stessa chiamata di funzione avrà effetti differenti, in base all'oggetto che la riceve.
- **Coesistenza con i sistemi esistenti.** La separazione della definizione di un oggetto dalla sua implementazione è perfetta per incapsulare le applicazioni esistenti. Usando l'IDL, si può trasformare il codice esistente come se fosse un oggetto visibile all'ORB.

3.2.1 L'ORB in dettaglio

Un ORB CORBA è un componente software che si occupa delle relazioni tra i client e i server. Usando un ORB, il client può trasparentemente invocare i metodi di un oggetto server che può stare sulla stessa macchina o distribuito su una rete. L'ORB intercetta le chiamate ed è responsabile di cercare un oggetto che è in grado d'implementare la richiesta, di passargli i parametri, invocare il metodo e restituire il risultato. Il client non è a conoscenza della locazione dell'oggetto, del suo linguaggio di implementazione, del suo Sistema Operativo, o di qualsiasi altro aspetto che non è stato incluso nell'interfaccia. È molto importante notare che le regole attinenti all'architettura client/server sono usate solo per coordinare le interazioni tra due oggetti. Infatti, gli oggetti di un ORB possono agire sia da client che da server, dipende dall'occasione.

La Figura 3.2 mostra l'architettura CORBA in dettaglio ed evidenzia tutti i suoi componenti tipici. Nonostante la sua complessità, non è poi tanto complicata da capire come potrebbe apparire ad una visione superficiale. La chiave di lettura è capire che CORBA fornisce una meccanismo sia statico che dinamico per l'invocazione dei metodi di un'interfaccia.

Spieghiamo adesso a cosa servono i componenti dal lato client:

- **I client IDL stub** forniscono un'interfaccia statica per i servizi che offrono gli oggetti. Dal punto di vista del client, lo **stub** agisce come una chiamata locale, è un *proxy* locale per un oggetto remoto. I servizi sono definiti usando l'IDL, e sia gli

stub del client che quelli del server sono generati automaticamente dal compilatore IDL. Un client deve avere uno stub IDL per ogni interfaccia del server che usa. Lo stub include il codice per il marshaling dei parametri e dei valori di ritorno. Questo significa che codifica e decodifica le operazioni e i loro parametri per spedirle al server. Includono inoltre degli *header file* che permettono di invocare i metodi del server da un linguaggio ad alto livello come ad esempio il C, C++, Java, senza preoccuparsi delle convenzioni di chiamata di una funzione e del marshaling. Basta semplicemente invocare una normale funzione all'interno del programma per ottenere un servizio remoto.

- **La Dynamic Invocation Interface (DII)** permette di scoprire i metodi che possono essere invocati a tempo di esecuzione. CORBA definisce della API standard che possono essere chiamate per ricevere i *metadati* che descrivono le interfacce dei server.
- **Le API dell'Interface Repository** permettono di ottenere e modificare la descrizione di tutte le interfacce dei componenti registrati, i metodi che supportano e i parametri che richiedono. La specifica CORBA chiama questa descrizione *method signature.* L'Interface Repository non è altro che un database distribuito che contiene una descrizione, comprensibile per una macchina, delle interfacce definite con il linguaggio IDL. Le API permettono ai componenti di accedere, memorizzare e aggiornare dinamicamente i *metadati.* Questo uso dei *metadati* permette ad ogni componente di avere un'interfaccia che descrive tutte le altre sue interfacce. L'ORB stesso È realizzato in questo modo.
- **L'interfaccia dell'ORB** consiste di alcune API che forniscono dei servizi locali che possono essere utili ad un'applicazione. Per esempio, CORBA fornisce delle funzioni per convertire un riferimento ad un oggetto in una stringa e viceversa. Questi servizi possono essere molto utili per inviare o memorizzare un riferimento ad un oggetto.

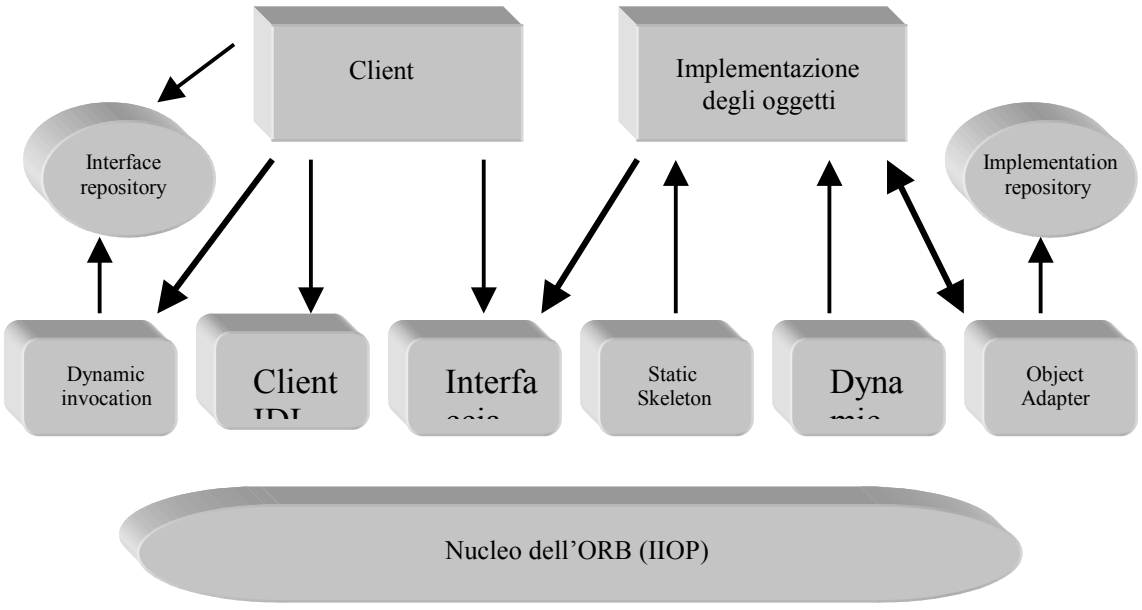


Figura 3.2 Schema dell'architettura CORBA e di tutti i suoi componenti.

Il supporto delle chiamate sia statiche che dinamiche, così come l'Interface Repository, rende CORBA l'architettura client/server più sofisticata che offre attualmente il mercato.

Le invocazioni statiche sono facili da programmare, veloci e autodocumentanti. Le invocazioni dinamiche forniscono la massima flessibilità, ma sono difficili da programmare e hanno delle prestazioni scadenti. Comunque sono molto utili per quei **tool** che generano codice scoprendo i servizi che un componente offre.

Dal lato server non si può dire se una chiamata è statica o dinamica. Entrambe hanno la stessa semantica. In entrambi i casi, l'ORB localizza un server, trasmette i parametri e trasferisce il controllo all'implementazione dell'oggetto attraverso lo **stub** del server (viene anche chiamato **skeleton** per distinguerlo da quello del client).

Vediamo adesso i componenti che si trovano sul lato server:

- **I server IDL stub (o skeleton)** forniscono un'interfaccia statica ad ogni servizio esportato da un server. Questi **stub**, come quelli dal lato client, sono creati usando il compilatore IDL.
- **La Dynamic Skeleton Interface (DSI)**, introdotta a partire dalla versione 2.0 di CORBA, fornisce un meccanismo a tempo di esecuzione per far comunicare i client e i server senza aver bisogno né dello **stub** e né dello **skeleton**. Funzione più o meno in questo modo: quando le arriva un messaggio, guarda i parametri per capire a quale oggetto server è destinato e qual è il metodo richiesto. Al contrario gli skeleton sono definiti staticamente per una particolare classe di oggetti e presuppongono un'implementazione per ogni funzione definita nel file IDL. La Dynamic Skeleton Interface è molto utile per implementare dei generici metodi di comunicazione tra i vari ORB. Può anche essere usata dagli interpreti a dai linguaggi di scripting per generare dinamicamente le implementazioni degli oggetti. La DSI è l'equivalente dal lato server della DII vista in precedenza. Essa può ricevere delle invocazioni sia statiche che dinamiche.
- **L'Object Adapter** si trova al centro dei servizi di comunicazione forniti dall'ORB e accetta le richieste dei servizi per conto degli oggetti server. Fornisce l'ambiente a tempo di esecuzione per istanziare gli oggetti server, per passargli le richieste e per assegnargli un identificatore di oggetto (**ID**) che CORBA chiama **object reference** o meglio riferimento all'oggetto. L'Object Adapter registra anche le classi che supporta e le loro istanze a tempo di esecuzione (l'implementazione) servendosi dell'**Implementation Repository**. CORBA specifica che ogni ORB deve supportare un **adapter** standard chiamato **Basic Object Adapter** o **BOA**. A partire dalla versione 2.2 e stato specificato anche un **adapter** molto più funzionale, il **Portable Object Adapter** o **POA**. Comunque i server possono supportare anche degli adapter proprietari oltre ovviamente a quelli standard.
- **L'Implementation Repository** fornisce un database a tempo di esecuzione per le informazioni attinenti alle classi che un server supporta, gli oggetti che sono stati

istanziati e i loro ID. Serve anche come posto per memorizzare delle informazioni addizionali associate all'implementazione dell'ORB.
- **L'interfaccia dell'ORB** consiste di alcune API per dei servizi locali che sono identiche a quelle che abbiamo visto sul lato client.

3.3 I Repository ID

A partire dalla versione 2.0 di CORBA, gli ORB forniscono degli identificatori globali, chiamati **Repository ID**, per identificare globalmente ed unicamente un componente e le sue interfacce tra i vari ORB e le varie **repository**. I Repository ID, generati da algoritmi o forniti dall'utente, sono delle stringhe uniche che servono a mantenere una consistenza nelle convenzioni dei nomi usate dalle **repository** fornite dai diversi produttori. Non sono permesse collisioni di nomi o ambiguità. Sono generati tramite delle direttive *pragma* che sono inserite all'interno del file IDL. Queste direttive specificano se l'ID è un **DCE Universal Unique Identifiers** (**UUID**) come quelli visti per COM, oppure se è un nome fornito dall'utente che segue le convenzioni di scope del file IDL. Vediamoli più in dettaglio:

- **DCE Universal Unique Identifiers (UUID).** È molto simile a quello che abbiamo visto per COM. La **DCE** fornisce un generatore di **UUID** che calcola un numero globalmente unico usando la data e l'ora corrente, la **network card ID** ed un contatore ad alta frequenza. Non c'è nessuna possibilità di creare due ID uguali. Il formato DCE per il Repository ID consiste di tre parti separate dai due punti (":"). La prima parte è la stringa *"DCE"*. La seconda è l'UUID in un formato stampabile. La terza consiste in un numero di versione in formato decimale. Per esempio potremmo avere una cosa del genere: *"DCE:700dc500-0111-22ce-aa9f:1"*.
- **Nomi IDL definiti dall'utente.** Si possono creare dei Repository ID usando dei nomi IDL e sono formati anche questi da tre parti separate dai due punti (":"). La prima parte è la stringa *"IDL"*. La seconda è una lista di identificatori separati dal carattere "/". Il primo identificatore è un prefisso unico (ad esempio il nome del produttore del componente), mentre gli altri sono dei nomi che seguono le regole di *scope* del file IDL. La terza parte consiste di due numeri di versione in formato decimale separati da un punto ".". Ad esempio, un Repository ID valido per l'interfaccia MyInterface che si trova nel modulo MyModule che è stato creato dal produttore MyFirm è: *"IDL:MyFirm/MyModule/MyInterface"*. In questo caso il prefisso unico è il nome del produttore, ma può essere qualsiasi nome unico come ad esempio un ID di Internet.

Le direttive *pragma* indicano al precompilatore IDL come generare i Repository ID e come associarli ad una particolare interfaccia. CORBA ne definisce tre tipi:

- La **prefix pragma** setta un prefisso che sarà appeso a tutti i Repository ID definiti successivamente, finché non verrà specificato un nuovo prefisso. Il formato di

questa *pragma* è: *"#pragma prefix <stringa>"*. Ad esempio potremmo avere: *"pragma prefix MyFirm"*.

- La **version pragma** appende un numero di versione ad un Repository ID per un nome IDL specifico. Il formato di questa direttiva è: *"#pragma version <nome> <maggiore> .<minore>"*. Se non si usa questa direttiva si suppone che la versione sia la 1.0. Vediamo anche qui un esempio: *"#pragma version MyInterface 1.3"*.
- La **ID pragma** associa un arbitrario Repository ID ad un nome specifico IDL. Il formato di questa *pragma* è: "#pragma ID <name> <id>". Un esempio è il seguente: *"#pragma ID MyInterface DCE:700dc500-0111-22ce-aa9f:1"*.

3.4 Gli Object Reference

Gli **Object Reference** o riferimenti agli oggetti forniscono tutte le informazioni di cui si ha bisogno per specificare unicamente un oggetto all'interno di un ORB. Sono dei nomi unici o identificatori. L'implementazione degli Object Reference non è definita dalla specifica CORBA, quindi significa che ogni produttore ha fornito una sua soluzione proprietaria; in altre parole due diversi ORB possono avere una diversa rappresentazione dei riferimenti agli oggetti. Con la versione 2.0 sono stati introdotti gli **Interoperable Object References** (**IOR**) che i diversi produttori devono supportare per poter trasmettere i riferimenti agli oggetti tra ORB diversi.

Il metodo escogitato per avere una certa portabilità degli Object Reference è quello di obbligare gli ORB a fornire lo stesso mapping dei Reference per un particolare linguaggio di programmazione. Questo significa che è il linguaggio *target* a fornire la portabilità e a permettere ai riferimenti agli oggetti di poter funzionare con differenti ORB all'interno dello stesso programma. Cosa succede se un programma usa gli Object Reference su due diversi ORB? In base alla specifica CORBA il programma dovrebbe continuare a funzionare correttamente. Infatti, il problema di evitare eventuali conflitti sugli oggetti è a carico dei produttori degli ORB, grazie appunto agli IOR.

I client ottengono usualmente gli Object Reference invocando dei servizi su altri oggetti dei quali hanno un Reference, oppure da particolari elenchi in fase di inizializzazione (ci occuperemo di questo in seguito).

Si possono convertire in stringhe per poterli successivamente memorizzare in file. Queste stringhe possono essere conservate o trasmesse in differenti maniere e possono essere riconvertite in Reference. Per questo scopo CORBA definisce due funzioni, **object_to_strin**g e **string_to_object** che fanno parte dell'interfaccia dell'ORB. I programmi client possono usere queste due funzioni per ottenere una stringa e convertirla in un Reference, o viceversa.

L'interfaccia dell'ORB definisce, inoltre, altre funzioni che possono essere invocate su qualsiasi Object Reference. Queste operazioni sono direttamente implementate

dall'ORB e, quindi, quando le si invoca non si passa per l'implementazione dell'oggetto. Si può ad esempio invocare il metodo **get_interface** su qualsiasi Reference per ottenere un oggetto Interface Repository ed ottenere da quest'ultimo delle informazioni sul tipo dell'oggetto, oppure si può usare la funzione **get_implementation** per ricevere un oggetto Implementation Repository da cui avere delle informazioni sull'implementazione dell'oggetto. Un altro esempio è la funzione **is_nil** che controlla se un oggetto esiste.

3.5 L'Interface Repository

L'**Interface Repository** è un database *on-line* di definizioni di oggetti. Queste informazioni vengono generate direttamente dal compilatore IDL oppure dalle funzioni che servono per scrivere nell'Interface Repository. La specifica CORBA non si preoccupa della creazione di queste informazioni, ma definisce dettagliatamente come devono essere organizzate e accedute. Inoltre, suggerisce una serie di classi le cui istanze servono per rappresentare gli oggetti specificati con il linguaggio IDL. Il risultato finale di tutto ciò è un database di oggetti molto flessibile che mantiene la stessa gerarchia dell'IDL, ma che contiene informazioni in forma compilata e non in forma di sorgente.

Un ORB ha bisogno di capire la definizione degli oggetti con cui sta lavorando. Un modo per farlo è incorporare questa definizione nei file **stub**. L'alternativa è ottenere le informazioni dinamicamente accedendo all'Interface Repository. Una volta che l'ORB ha ricevuto i dati li può usare per:

- Fornire il controllo dei tipi sui metodi. Il tipo dei parametri dei metodi viene controllato indipendentemente dal fatto che la chiamata sia statica o dinamica.
- Aiutare la connessione tra ORB. Le informazioni dell'Interface Repository si possono usare per trasmettere oggetti tra ORB diversi. In questo caso è necessario usare lo stesso identificatore (**Repository ID**) per individuare univocamente gli oggetti condivisi.
- Fornire delle informazioni (*metadati*) ai client e ai **tool**. I client usano l'Interface Repository per creare dinamicamente le invocazioni dei metodi. I **tool**, come ad esempio i **browser** di classi, i generatori di applicazioni e i compilatori, possono usare queste informazioni per ottenere la struttura e la definizione delle classi a tempo di esecuzione.
- Fornire degli oggetti autodescriventi. Si può invocare il metodo **get_interface** su qualsiasi oggetto CORBA per ottenere le informazioni sull'interfaccia che supporta (sempre che queste siano state memorizzate nell'Interface Repository).

Un Interface Repository può essere mantenuta localmente o può essere gestita come una risorsa distribuita, l'importante è che fornisca dei dati sulla struttura delle classi e sulle interfacce degli oggetti in essa registrati. Un ORB può avere accesso a diverse Interface Repository.

Un'Interface Repository è implementata come un insieme di oggetti che rappresentano le informazioni che contiene. Questi oggetti devono essere persistenti, cioè devono essere memorizzati su un supporto non volatile. CORBA raggruppa i *metadati* in moduli che rappresentano lo spazio dei nomi. I nomi devono essere unici all'interno dei moduli.

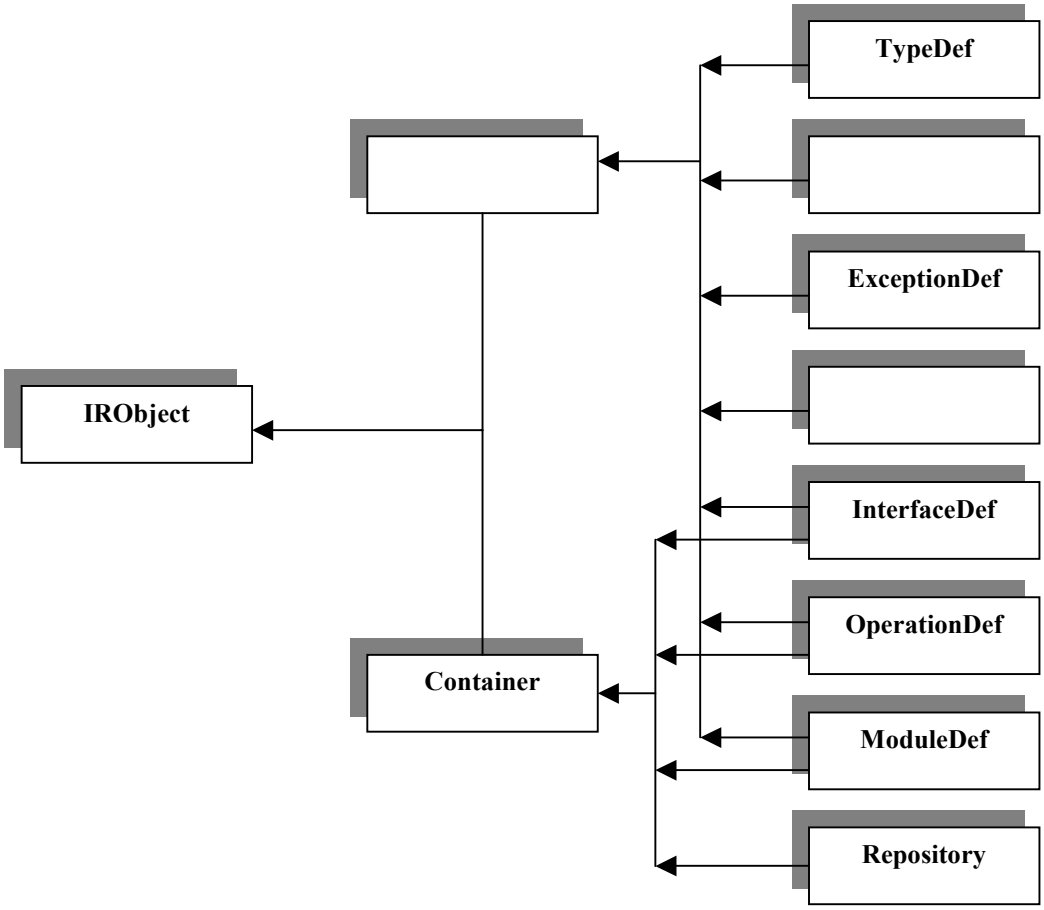


Figura 3.3 Gerarchia delle classi contenute nell'Interface Repository.

CORBA definisce un'interfaccia per ogni costrutto IDL:

- **ModuleDef** definisce un gruppo logico di interfacce. Come un file IDL, l'Interface Repository usa i moduli per raggruppare le interfacce e per *navigare* all'interno dei gruppi in base al loro nome. Si può pensare ad un modulo come se fosse uno spazio dei nomi.
- **InterfaceDef** definisce le interfacce degli oggetti. Contiene le costanti, le definizioni di tipo, le eccezioni e le definizioni tipiche dell'interfaccia, cioÈ gli attributi e i metodi.
- **OperationDef** definisce un metodo di un'interfaccia di un oggetto. Contiene la lista dei parametri e le eccezioni che il metodo può sollevare.

- **ParameterDef** definisce un argomento di un metodo.
- **AttributeDef** definisce un attributo di un'interfaccia.
- **ConstantDef** definisce una costante con nome.
- **ExceptionDef** definisce un'eccezione che può essere sollevata da un'operazione.
- **TypeDef** definisce un tipo che è parte di una definizione IDL.

Oltre a queste otto interfacce, CORBA specifica l'interfaccia **Repository,** che serve come radice per tutti i moduli contenuti in un'Interface Repository, e le classi astratte **IRObject**, **Contained** e **Container**. L'ereditarietà fra questi oggetti è mostrata nella Figura 3.3.

Tutti gli oggetti dell'Interface Repository ereditano dall'interfaccia **IRObject**, che è stata introdotta dalla versione 2.0 di CORBA. Questa interfaccia fornisce un attributo per identificare il tipo dell'oggetto e un metodo per distruggerlo.

Gli oggetti che sono contenitori ereditano le operazioni necessarie per esaminare il loro contenuto dall'interfaccia **Container**. Invece, l'interfaccia **Contained** definisce il comportamento di quegli oggetti che sono contenuti in altri oggetti. Quindi, tutte le classi che descrivono gli oggetti dell'Interface Repository sono derivate da **Container**, da **Contained** o da entrambi attraverso l'ereditarietà multipla. Questo ingegnoso schema di gerarchia permette agli oggetti di comportarsi in base alle loro relazioni di contenimento.

Le classi dell'Interface Repository forniscono quelle operazioni che permettono di leggere, scrivere e distruggere i *metadati* in essa contenute. Ad esempio, il metodo **destroy** elimina un oggetto dal Repository; se si chiama quest'operazione su un contenitore (**Conteiner**), tutto il suo contenuto sarà cancellato. L'interfaccia **Contained** fornisce, fra gli altri, il metodo **move** che rimuove un oggetto da un contenitore e lo sposta in un altro.

Si può accedere ai *metadati* dell'Interface Repository invocando polimorficamente i metodi dei suoi oggetti. Con lo schema dell'ereditarietà visto in precedenza, si possono scorrere ed estrarre informazioni dagli oggetti del Repository con solamente nove metodi, cinque dei quali appartengono alle classi madri **Container** e **Contained**. Gli altri quattro metodi sono specifici delle interfacce **InterfaceDef** e **Repository**. Vediamo la descrizione di questi metodi:

- **Describe**. Quando si invoca questo metodo su un oggetto **Contained**, restituisce un struttura **Description** che contiene le informazioni IDL che descrivono l'oggetto.
- **Lookup**. Quando si invoca questo metodo su un oggetto **Container**, restituisce una sequenza di puntatori agli oggetti che contiene.
- **Lookup_name**. Si invoca questo metodo su un oggetto **Container** per localizzare un oggetto in base al nome.
- **Contents**. Quando si invoca questo metodo su un oggetto **Container**, restituisce una lista di oggetti direttamente contenuti o ereditati da questo oggetto. Si usa questo metodo per scorrere una gerarchia di oggetti. Per esempio, si può partire da

un oggetto Repository, elencare tutti gli oggetti che contiene e proseguire su uno di essi.

- **Describe_contents**. Si invoca questo metodo su un oggetto **Container** e restituisce una sequenza di puntatori alla descrizione del contenuto degli oggetti che contiene. Questa funzione combina le operazioni **contents** e **describe**. Si può limitare la ricerca escludendo gli oggetti ereditati o cercando un tipo particolare di oggetto, per esempio un **InterfaceDef**.
- **Describe_interface**. Quando si invoca questo metodo su un oggetto **InterfaceDef**, restituisce una struttura che descrive completamente l'interfaccia, incluso il suo nome, il **Repository ID**, il numero di versione, le operazioni, gli attributi e tutte le interfacce da cui eredita.
- **Is_a**. Si invoca questo metodo su un oggetto **InterfaceDef** e restituisce *TRUE* se l'interfaccia è identica o eredita direttamente dall'interfaccia che viene fornita come parametro.
- **Lookup_id**. Si invoca questo metodo su un oggetto **Repository** per cercare un oggetto al suo interno in base al **Repository ID**.

Queste chiamate di metodi permettono di *navigare* attraverso il contenuto dell'Interface Repository. Si può scorrere attraverso lo spazio dei nomi o specificando un modulo in cui cercare un oggetto che soddisfa certi requisiti. Quando l'oggetto è stato trovato, si usa il metodo **describe** per ricevere le informazioni IDL che lo descrivono.

È importante ricordare che **IRObject**, **Container** e **Contained** sono classi astratte, il che significa che non si possono mai gestire direttamente. Per invocare i loro metodi si devono prima derivare delle classi concrete.

Si può ottenere l'interfaccia di un oggetto in tre modi differenti:

1. Chiamando direttamente il metodo **Object::get_interface**. Si può invocare il metodo su un qualsiasi riferimento ad oggetto valido. La chiamata restituisce un oggetto **InterfaceDef**. Questo metodo è utile quando si incontra un oggetto di cui non si conosce il tipo a tempo di compilazione.
2. Cercando per nome all'interno di uno spazio dei nomi. Per esempio, se si conosce il nome dell'interfaccia che si sta cercando, si può iniziare la ricerca nel modulo radice dell'Interface Repository. Una volta che la voce è stata trovata, si può invocare il metodo **InterfaceDef::describe_interface** per ottenere i *metadati* che descrivono l'interfaccia.
3. Localizzando l'oggetto **InterfaceDef** che corrisponde ad un particolare **Repository ID**. Si può fare ciò invocando il metodo **Repository::lookup_id**.

Una volta che si è ottenuto l'oggetto **InterfaceDef**, si può usare la sua interfaccia per ottenere i *metadati* di cui si ha bisogno e, con questi, creare un'invocazione dinamica di un metodo.

Con le migliorie apportate a partire dalla versione 2.0 della Specifica CORBA, si possono avere molte Interface Repository che operano attraverso diversi ORB. Per evitare le collisioni dei nomi si assegnano dei Repository ID alle interfacce e alle operazioni per determinarle univocamente.

Si possono usare i Repository ID per replicare delle copie dei *metadati* nelle diverse Interface Repository in modo tale da mantenere la coerenza fra di loro. I questo modo l'identità unica di un'interfaccia sarà conservata anche oltre i confini delle Repository e degli ORB. Per esempio con un Repository ID globale si possono ottenere delle informazioni dall'Interface Repository locale e dei *metadati* addizionali, relativi alla stessa interfaccia, da un'Interface Repository remota o addirittura di un diverso ORB.

In Mico l'Interface Repository è implementata da un programma separato che si chiama **ird**. L'idea è di eseguire una sola istanza di questo programma per tutte le applicazioni Mico. Con l'opzione da riga di comando **–ORBIfaceRepoAddr** si indica ai programmi quale Interface Repository usare. Ma come si ottiene l'indirizzo del programma **ird**? La soluzione è quella di specificare, con l'opzione **–ORBIIOPAddr**, a quale indirizzo l'**ird** si deve collegare. Spiegheremo meglio questi concetti successivamente. Ecco un esempio:

```
ird –ORBIIOPAddr inet:<nome host ird>:8888
```

dove <*nome host ird*> deve essere rimpiazzato con il nome dell'*host* sul quale è in esecuzione il programma **ird**. Dopodiché le applicazioni Mico devono essere fatte partire in questo modo:

```
applicazione –ORBIfaceRepoAddr inet:<nome host ird>:8888
```

dove, ovviamente, *applicazione* è il nome del programma che si vuole legare a quella particolare Interface Repository.

Per evitare di dover scrivere delle righe di comando così lunghe, si possono inserire le opzioni nel file *.micorc*.

Il programma **ird** accetta i seguenti argomenti da riga di comando:

- **--help**. Mostra una lista di tutte le opzioni da riga di comando ed esce.
- **--db <file database>**. Specifica il nome del file in cui l'**ird** deve salvare il contenuto dell'Interface Repository quando termina. Quando viene fatto ripartire, leggerà il file specificato con questa opzione per ripristinarne il contenuto. È da notare che il contenuto del database è costituito da caratteri ASCII che rappresentano i costrutti IDL.

3.6 L'interfaccia per l'inizializzazione

CORBA definisce un insieme di metodi per l'inizializzazione che tutti gli ORB devono fornire per aiutare un oggetto a cavarsela da solo in un ambiente distribuito. Questi metodi sono implementati dallo pseudo oggetto **CORBA::ORB**, che fa parte del nucleo dell'ORB. I metodi relativi all'inizializzazione sono in particolare **BOA_init**, **list_initial_services** e **resolve_initial_references**.

Si deve invocare **BOA_init** per ottenere un riferimento allo pseudo oggetto **BOA**. Questo riferimento è necessario per registrare i propri oggetti con l'ORB. Il metodo **list_initial_services** serve per ottenere una lista di nomi di servizi forniti dall'ORB. È come una specie di **Naming Service**. Il metodo **resolve_initial_references** converte il nome dei servizi in riferimenti ad un oggetto.

In essenza, vediamo i passi che si devono eseguire per permettere ad un oggetto di trovare i servizi di cui ha bisogno per funzionare con un ORB:

1. **Ottenere un Object Reference dell'ORB**. Con la chiamata **ORB_init**, un oggetto informa l'ORB della propria presenza e ottiene un riferimento ad esso. È da notare che questa funzione è una API e non un invocazione di metodo. Infatti, prima di poter invocare un metodo, si deve accedere all'ambiente distribuito CORBA.
2. **Ottenere un puntatore all'Object Adapter**. Si deve invocare il metodo **BOA_init** sullo pseudo oggetto ORB per ottenere un suo riferimento. Anche BOA è uno pseudo oggetto.
3. **Scoprire quali servizi iniziali sono disponibili**. Il metodo **list_initial_services** serve appunto per questo scopo. La chiamata restituisce un elenco di nomi che rappresentano i servizi forniti dall'ORB, come ad esempio l'**Interface Repository**, il **Trader**, il **Naming Service**, ecc. Parleremo dei servizi successivamente.
4. **Ottenere un riferimento per il servizio che si vuole usare**. Con il metodo **resolve_initial_reference** si ottiene a partire dal nome di un servizio un riferimento ad esso.

È da notare che un oggetto può inizializzarsi in più di un ORB. Una volta ottenuto un riferimento ad un servizio, lo si può sfruttare per cercare e collegare con gli altri oggetti raggiungibili tramite quell'ORB. Ad esempio con il Naming Service si possono cercare gli oggetti in base al loro nome.

Vediamo adesso un esempio di inizializzazione con Mico, dove l'ORB è implementato come una libreria (**libmico2.2.7.a**) che è *linkata* alle applicazioni. Tutte le applicazioni Mico devono chiamare la funzione di inizializzazione dell'ORB **ORB_init** prima di poter usare qualsiasi altro metodo remoto.

```
int main (int argc, char *argv[])
{
   CORBA::ORB_var orb = CORBA::ORB_init(argc, argv,
      "mico-local-orb");
   CORBA::BOA_var boa = orb->BOA_init(argc, argv,
      "mico-local-boa");
   ...
```

}

In questo modo, cioè passando come argomenti *argc* e *argv*, si da la possibilità all'ORB di accedere agli argomenti della riga di comando. Il terzo parametro indica l'ORB a cui ci vogliamo collegare, ma per il momento si può usare solamente il valore specificato nell'esempio. Le variabili che finiscono per *"_var"* sono degli *smart pointer*.

L'ORB rimuove dalla riga di comando i parametri che comprende. Gli stessi parametri possono essergli passati anche attraverso un file, chiamato *".micorc"*, che l'ORB legge quando viene inizializzato. I parametri della riga di comando hanno la priorità rispetto a quelli indicati nel file.

Ecco una breve descrizione dei principali parametri che possono essere passati all'ORB:

- **-ORBIIOPAddr <indirizzo>**. Setta l'indirizzo IIOP che il server deve usare. Spiegheremo il suo uso successivamente. Se non si usa questa opzione, il server sceglierà un indirizzo IIOP libero. Si può usare questa opzione più di una volta per permettere al server di rimanere in ascolto su più indirizzi.
- **-ORBId <identificatore ORB>**. Specifica l'identificatore dell'ORB. Attualmente è supportato solo il valore *mico-local-orb*. Questa opzione è stata prevista per quei programmi che hanno bisogno di accedere a diverse implementazioni di CORBA nello stesso processo.
- **-ORBImplRepoAddr <indirizzo Implementation Repository>**. Specifica l'indirizzo del processo su cui è in esecuzione l'Implementation Repository. L'ORB cercherà di collegarsi con l'oggetto specificato a questo indirizzo. Se la connessione fallisce, l'ORB farà partire una sua Repository locale.
- **-ORBImplRepoIOR <IOR Implementation Repository>**. È identico al parametro precedente, solamente che qui l'indirizzo è specificato in forma di **Interoperable Object Reference** o **IOR**.
- **-ORBIfaceRepoAddr <indirizzo Interface Repository>**. Funziona come il parametro **–ORBImplRepoAddr**, ma specifica invece l'indirizzo dell'Interface Repository.
- **-ORBIfaceRepoIOR <IOR Interface Repository>**. È identico al parametro precedente, ma qui l'indirizzo è un **IOR**.
- **-ORBNamingAddr <indirizzo Naming Service>**. Specifica l'indirizzo del processo su cui è in esecuzione il **Naming Service**.
- **-ORBNamingIOR <IOR Naming Service>**. È identico al parametro precedente, ma qui l'indirizzo è un **IOR**.
- **-ORBInitRef <identificatore> = <IOR>**. Setta il valore <*IOR*> che verrà restituito quando si chiamerà il metodo **resolve_initial_references** con il parametro <*identificatore*>.
- **-ORBDefaultInitRef <lista IOR>**. Setta una lista di servizi che saranno restituiti per *default* quando si invocherà il metodo **resolve_initial_references**.

- **-ORBConfFile <rcfile>**. Specifica il nome del file dal quale leggere gli eventuali parametri addizionali. Per default si usa il file *".micorc"*.
- **-ORBBindAddr <indirizzo>**. Specifica un indirizzo al quale la funzione **bind** cercherà di collegarsi. Questa opzione può essere usata più di una volta per specificare molti indirizzi.

Segue nell'esempio l'inizializzazione del BOA. In Mico il BOA è implementato parzialmente con la libreria **libmico2.2.7.a** e con un programma separato (**micod**) che è chiamato **BOA daemon**.

Anche qui si passano al BOA i parametri della riga di comando, che vengono valutati ed eventualmente rimossi. Quelli principali sono:

- **-OAId <identificatore BOA>**. Specifica l'identificatore del BOA. Attualmente solo *mico-local-boa* è supportato.
- **-OAImplName <nome dell'implementazione dell'oggetto>**. Indica al server il nome della sua implementazione. Questa opzione deve essere usata quando si mandano in esecuzione dei server persistenti che devono essere registrati col BOA daemon.
- **-OARemoteAddr <indirizzo remoto BOA>**. Questa opzione serve per indicare all'implementazione di un oggetto l'indirizzo del BOA daemon. Si usa quando si fanno partire dei server persistenti che devono essere registrati.

Il metodo **list_initial_services** si usa per avere la lista dei servizi di cui si può ottenere un riferimento dall'ORB. Ecco come si usa:

```
int main (int argc, char *argv[])
{
   CORBA::ORB_var orb = CORBA::ORB_init(argc, argv,
      "mico-local-orb");
   ...
   CORBA::ORB::ObjectIdList_var ids =
      orb->list_initial_services();

   for (int i = 0; i < ids->length(); i++)
      cout << ids[i] << endl;
   ...
}
```

I riferimenti iniziali possono essere specificati con l'opzione **–ORBInitRef** e **–ORBDefaultInitRef** che abbiamo visto in precedenza.

Vediamo adesso un esempio che mostra come ottenere un Object Reference per l'Interface Repository usando il metodo **resolve_initial_references**:

```
int main (int argc, char *argv[])
{
```

```
    CORBA::ORB_var orb = CORBA::ORB_init(argc, argv,
       "mico-local-orb");
    ...
    CORBA::Object_var obj =
       orb->resolve_initial_references("InterfaceRepository");
    CORBA::Repository_var repo =
       CORBA::Repository::_narrow(obj);
    ...
}
```

Il metodo **_narrow** esegue semplicemente il *cast* da un oggetto **Object** ad uno **Repository**.

Se si specifica l'Interface Repository usando il parametro **–ORBIfaceRepoAddr** oppure **–ORBIfaceRepoIOR**, il riferimento restituito da **resolve_initial_references** sarà quello specificato. Altrimenti, l'ORB farà partire una sua Interface Repository locale e restituirà un riferimento ad essa.

Per ottenere un riferimento all'Implementation Repository oppure al Naming Service si usa un meccanismo analogo; però, in questo caso gli identificatori da usare saranno rispettivamente *"ImplementatioRepository"* e *"NameService"*.

3.7 Il Mico Binder

Come si può ottenere un Object Reference per l'Interface Repository, l'Implementation Repository e il Naming Service, soprattutto se i client risiedono su macchine diverse? Nel paragrafo precedente abbiamo visto come usare il metodo **resolve_initial_references**, ma questo non fa altro che spostare il problema dai client all'ORB.

Siccome lo standard CORBA non offre una soluzione per questo problema, Mico ha inventato un suo metodo. È da notare, però, che usando questa caratteristica proprietaria di Mico, i programmi diventano incompatibili con le altre implementazioni di CORBA.

Il **Mico Binder** è funziona più o meno come il Naming Service, ma mappa la coppia (*Indirizzo*, *Repository ID*) con gli Object Reference. Come abbiamo già detto in precedenza un *Repository ID* è una stringa che identifica un oggetto IDL, mentre un *indirizzo* identifica un processo su un computer. Attualmente Mico definisce tre tipi di indirizzi: un *idirizzo internet*, un *indirizzo unix* e un *indirizzo locale*. Un *indirizzo internet* è una stringa che ha questo formato:

```
inet:<nome host>:<numero porta>
```

che si riferisce al processo sulla macchina <*nome host*> la cui porta TCP è <*numero porta*>. Un *indirizzo unix* è del tipo:

```
unix:<nome socket>
```

e si riferisce al processo sulla macchina corrente che ha il **socket unix** legato a <*nome socket*>. Infine, un indirizzo locale è del tipo:

```
local:
```

e si riferisce al processo corrente sulla macchina corrente, come ad esempio il processo *this*.

Vediamo un esempio. Supponiamo di avere un server che implementa, per il momento non ci interessa come, un'interfaccia IDL che ha questo Repository ID:

```
IDL:Account:1.0
```

e che ha questa definizione:

```
interface Account
{
   void deposit (in unsigned long amount);
   void withdraw (in unsigned long amount);
   long balance ();
};
```

Facciamo partire il server indicandogli l'*indirizzo internet*, che deve usare, con il comando (supponendo che il programma si chiami *server*):

```
server –ORBIIOPAddr inet:localhost:8888
```

Passiamo adesso al codice del client:

```
#include "account.h"

int main (int argc, char *argv[])
{
   // Inizializzazione dell'ORB
   CORBA::ORB_var =
      CORBA::ORB_init(argc, argv, "mico-local-orb");
   CORBA::BOA_var =
      orb->BOA_init(argc, argv, "mico-local-boa);

   CORBA::Object_var obj =
      orb->bind("IDL:Account:1.0", "inet:localhost:8888");
   if (CORBA::is_nil(obj))
   {
      // Collegamento fallito
```

```
    }
    Account_var client = Account::_narrow(obj);

    client->deposit(700);
    client->withdraw(250);
    cout << "Bilancio : " << client->balance() << endl;

    return 0;
}
```

Dopo aver inizializzato l'ORB e il BOA, il client usa il metodo **bind** per collegarsi con l'oggetto il cui Repository ID è IDL:Account:1.0 e che è in esecuzione in un processo che è in *ascolto* sulla porta 8888.

Se un server offre diversi oggetti dello stesso tipo (che hanno per esempio lo stesso Repository ID) e un client vuole collegarsi ad uno in particolare, ha bisogno di un meccanismo per distinguerli. Per risolvere il problema, si assegna un identificatore ad ogni oggetto che viene creato nel server, e si aggiunge lo stesso identificatore come ulteriore parametro nella chiamata a **bind**. Si possono usare le funzioni **ORB::string_to_tag** e **ORB::tag_to_string** per convertire una stringa in un identificatore e viceversa.

Per evitare di inserire nel codice del client l'indirizzo del server, si può eliminare il secondo parametro della **bind** e specificare una lista di indirizzi, a cui il client tenterà di collegarsi, con l'opzione **–ORBBindAddr**. Supponendo che il programma si chiami *client*, ecco un esempio sul suo uso:

```
client –ORBBindAddr local: -ORBBindAddr inet:localhost:8888
```

In questo modo la **bind** cercherà prima di collegarsi con un oggetto *Account* dello stesso processo e, in caso negativo, con un processo sulla stessa macchina che è in *ascolto* sulla porta 8888.

3.8 L'attivazione degli oggetti

Occupiamoci adesso della politica di attivazione degli oggetti e di come interagiscono con il **Basic Object Adapter (BOA)** o il **Portable Object Adapter (POA)** per dare ai client l'illusione che ogni oggetto che conoscono sia sempre disponibile e in funzione.

CORBA non fornisce un comando esplicito per far partire i server, con l'unica eccezione di quando si creano gli oggetti tramite la **CORBA Factory**. In questo caso, è la **Factory** stessa che implicitamente si occupa dell'esecuzione dell'oggetto.

Tocca al lato server dell'ORB dare l'illusione che i suoi oggetti, che possono essere veramente tanti, siano sempre attivi e in esecuzione. Questa illusione rende il codice

del client molto semplice, ma sposta il peso dell'implementazione sulla parte del server. Ciò significa che i componenti devono essere in grado di cooperare con l'ORB quando partono e quando si arrestano, oppure quando è l'ORB stesso ad entrare in esecuzione. Inoltre, l'ORB deve essere in grado sia di far partire i server automaticamente, sia quando un client ne richiede un servizio.

L'implementazione del server deve inoltre interagire con il **Persistence Service** per salvare e ricaricare lo stato degli oggetti. Anche questo processo deve essere totalmente trasparente al client. Per esempio, un client potrebbe memorizzare un Object Reference in un database e successivamente riutilizzarlo, magari molto tempo dopo, ritrovando l'oggetto nello stesso stato in cui l'aveva lasciato. È compito dell'ORB e del server preoccuparsi del salvataggio dello stato dell'oggetto e della sua terminazione (con il conseguente scaricamento dalla memoria) per permettere anche agli altri oggetti di poter funzionare.

3.9 Il BOA e l'interfaccia CORBA::BOA

Il **Basic Object Adapter (BOA)** è uno pseudo oggetto. Esso fornisce quelle operazioni che sono necessarie alle implementazioni degli oggetti server. Inoltre, si interfaccia con il nucleo dell'ORB e con gli **skeleton** delle implementazioni con delle interfacce proprietarie dei vari distrubutori, che non sono state ancora definite dalla specifica CORBA.

La Figura 3.4 mostra i metodi che l'OMG ha definito per l'interfaccia **CORBA::BOA**. Si usa questa interfaccia per creare e distruggere gli Object Reference, e per richiedere o aggiornare le informazioni che il BOA mantiene per un riferimento ad un oggetto.

Il BOA mantiene un registro degli oggetti attivi e delle implementazioni che controlla. Si usano i metodi della sua interfaccia per comunicare con questo registro e per avvisare l'ORB della presenza degli oggetti server.

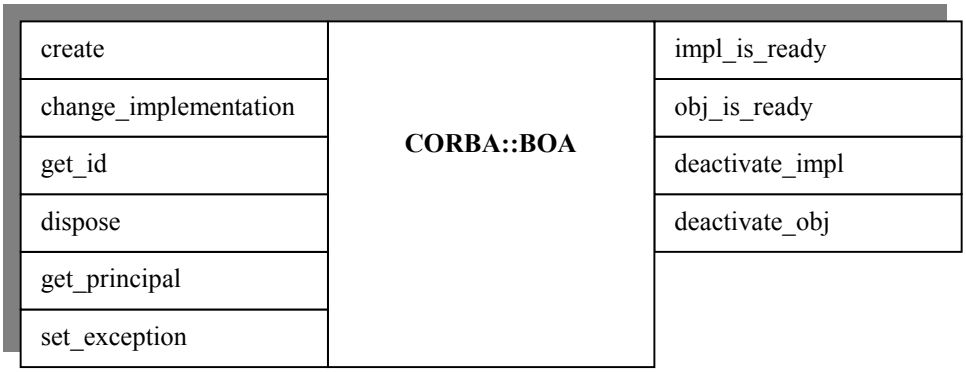


Figura 3.4 L'interfaccia CORBA::BOA.

Vediamo adesso una piccola descrizione di queste operazioni. Il metodo **create** è usato per descrivere al BOA l'implementazione di una nuova istanza di un oggetto e per ottenere un riferimento ad esso. L'oggetto si crea con il costruttore del linguaggio target o con una Factory. Si devono passare all'ORB tre tipi di informazioni per permettergli di legare un nuovo Object Reference ad un oggetto:

Il nome dell'interfaccia che è descritta nell'**Interface Repository**.
Il nome dell'implementazione che è descritta nell'**Implementation Repository**.
Gli **ID** per l'oggetto o **reference data** definiti dall'utente.

I **reference data** sono opachi per l'ORB e sono specifici di una determinata implementazione. Si usano per distinguere i diversi oggetti o per specificare un **Persistent ID** (**PID**) che indica il luogo in cui un oggetto memorizza il suo stato.

Si possono aggiornare le informazioni di un'implementazione associate ad un oggetto esistente invocando il metodo **change_implementation**. Con **get_id**, invece, si ottiene il reference data associato ad un oggetto. Per distruggere un riferimento ad un oggetto si usa la funzione **dispose** mentre, per distruggere l'oggetto stesso, si può usare il distruttore del linguaggio *target* o il **Life Cycle Service**.

Il metodo **get_principal** è obsoleto perché è stato rimpiazzato dal **Security Service**; deve comunque restituire un valore nullo (*null*). Per avvisare l'ORB che qualcosa non è andata per il verso giusto si usa la funzione **set_exception**.

Le ultime quattro operazioni, **impl_is_ready**, **obj_is_ready**, **deactivate_impl** e **deactivate_obj**, sono usate dai server e dall'ORB per attivare e disattivare gli oggetti che contengono. Ne parleremo successivamente.

Un Object Adapter, come abbiamo detto, definisce come un oggetto è attivato. Si può fare ciò creando un nuovo processo, creando un nuovo thread all'interno di un processo oppure riutilizzando un thread o un processo già esistente. Un server può supportare addirittura più di un Object Adapter per soddisfare diversi tipi di richieste. Comunque, l'OMG preferisce limitare la proliferazione di Object Adapter e, quindi, ha specificato il **Basic Object Adapter** e il **Portable Object Adapter** che tutti gli ORB devono supportare. Il primo fornisce i servizi essenziali di cui abbiamo già parlato, mentre il secondo è molto più evoluto e mette a disposizione anche dei servizi di portabilità.

CORBA richiede che le seguenti funzioni devono essere fornite da un'implementazione del BOA:

- Un Implementation Repository che permette di installare e registrare l'implementazione di un oggetto. Fornisce anche le informazioni necessarie per descrivere l'oggetto.
- Un meccanismo per generare e interpretare i riferimenti agli oggetti.
- Un meccanismo per attivare e disattivare le implementazioni degli oggetti.
- L'invocazione dei metodi e il passaggio dei loro parametri tramite gli skeleton.

CORBA fa una chiara distinzione tra server e gli oggetti in essi contenuti. Un server è un'unità in esecuzione, è un processo. Un oggetto, invece, implementa un'interfaccia. Un server può contenere uno o più oggetti (anche di differenti classi) o, all'altro estremo, può contenere il codice che implementa un singolo metodo invece di un'intera interfaccia. In ogni caso, gli oggetti vengono attivati dai server che li contengono.

Per avere la massima flessibilità, CORBA definisce cinque politiche di attivazione degli oggetti che sono: **shared server**, **unshared server**, **server per method**, **persistent server** e **library server**. Esse specificano le regole che una data implementazione deve seguire per attivare i suoi oggetti. I paragrafi successivi le descrivono in dettaglio.

Come abbiamo già detto in precedenza, in Mico il BOA è implementato parzialmente come una libreria e parzialmente con il programma **micod** (**BOA daemon**). Il programma **micod** è la parte del BOA che si occupa di far partire le implementazioni degli oggetti quando vengono richiesti i loro servizi e contiene, inoltre, l'Implementation Repository.

Per permettere alle applicazioni Mico di usare una singola Implementation Repository, si deve agire come abbiamo fatto per l'Interface Repository. Cioè, si deve indicare al programma **micod** l'indirizzo al quale si deve collegare con l'opzione **–ORBIIOPAddr** e indicare lo stesso indirizzo alle applicazioni con l'opzione **–ORBImplRepoAddr**. Per esempio:

```
micod –ORBIIOPAddr inet:<nome host micod>:9999
```

A questo punto le applicazioni possono essere fatte partire con:

```
applicazione –ORBImplRepoAddr inet:<nome host micod>:9999
```

Dove ovviamente *applicazione* è il nome di un programma Mico. Si può anche inserire l'opzione nel file *.micorc* e dare semplicemente il comando *applicazione*.

Il programma **micod** può accettare i seguenti parametri da riga di comando:

- **--help**. Mostra la lista di tutte le opzioni supportate ed esce.
- **--forward**. Indica a **micod** di usare il protocollo **GIOP** (**General Inter ORB Protocol**) che è molto più veloce e che non implica nessun *overhead*.

Sfortunatamente richiede delle proprietà particolari sul lato del client che non tutti gli ORB supportano (contrariamente a quando indicato dalla specifica CORBA). Questo è il motivo per cui l'opzione non è attivata per *default*.

- **--db <file database>**. Specifica il nome del file in cui il programma **micod** deve salvare il contenuto dell'Implementation Repository quando termina. Quando verrà fatto ripartire, leggerà il contenuto del file specificato con questa opzione per ripristinarne il contenuto.

L'Implementation Repository è il posto dove vengono memorizzate le informazioni sulle implementazioni degli oggetti. La specifica CORBA da solamente un'idea di cos'è l'Implementation Repository, ma non definisce la sua interfaccia (che quindi è specifica di ogni produttore).

Nell'implementazione di Mico, l'Implementation Repository contiene le seguenti informazioni di un server:

- Il nome che identifica unicamente il server.
- Il modo di attivazione che indica al BOA la politica di attivazione.
- La riga di comando o il percorso dove si trova il modulo. È la riga di comando che verrà eseguita dal BOA per far partire il server oppure, nel caso di un **server library**, è il percorso completo di dove si trova il modulo.
- La lista dei Repository ID per ogni interfaccia IDL implementata dal server.

Se abbiamo scritto un server che deve essere attivato automaticamente dal BOA quando si richiede un suo servizio, dobbiamo creargli una voce nell'Implementation Repository. Ciò si può fare usando il programma **ird**. Le sue principali funzionalità sono quelle di listare tutte le voci dell'Implementation Repository, mostrare delle informazioni dettagliate su una in particolare, creare una nuova voce oppure eliminarla.

Si deve specificare al programma **ird** l'Implementation Repository su cui deve operare con le solite opzioni **–ORBImplRepoAddr** o **–ORBImplRepoIOR**. Anche qui possiamo inserirle nel file *.micorc*.

Vediamo adesso qualche esempio sull'uso di **ird**. Per listare il contenuto dell'Implementation Repository si usa il comando:

```
imr list
```

Per avere delle informazioni dettagliate su una voce possiamo usare:

```
imr info <nome>
```

dove <*nome*> è il nome della voce. Per creare una voce si usa un comando un po' più complicato:

```
imr create <nome> <modo> <comando> <repID1> .. <repIDN>
```

dove <*nome*> è il nome della voce che si vuole creare, <*modo*> può essere *persistent*, *shared*, *unshared*, *permethod* e *library* che corrispondono alle politiche di attivazione, <*comando*> è la riga di comando che userà il BOA per attivare il server, e <*repID1*>, <*repIDN*> sono i Repository ID delle interfacce implementate dal server. Il comando:

```
imr delete <nome>
```

cancellerà la voce <*nome*>.

Registrare un'implementazione nell'Implementation Repository non significa attivarla. Usualmente, un server non persistente sarà attivato dal BOA solamente quando un client richiederà un suo servizio, ma può capitare di dover forzare l'attivazione di un'implementazione, per esempio per permetterle di registrarsi anche nel **Naming Service**. Per fare ciò si usa il comando:

```
imr activate <nome> [<indirizzo micod>]
```

che attiverà l'implementazione <*nome*>. In questo caso, il programma **imr** ha bisogno di sapere l'indirizzo del **BOA daemon**. Di solito questo è lo stesso indirizzo dell'Implementation Repository e quindi non c'è bisogno di specificarlo. Se però il **BOA daemon** è legato ad un indirizzo diverso da quello dell'Implementation Repository ed è diverso anche dall'indirizzo specificato con l'opzione **–ORBBindAddr**, lo si dovrà indicare.

Facciamo adesso un esempio completo per chiarire il tutto. Supponiamo di dover registrare il server che implementa l'interfaccia *Account* vista in precedenza e che né **micod**, né **ird** siano in esecuzione. Assumiamo, inoltre, che il nome dell'*host* sia *zirkon*.

Prima di tutto creiamo il file *.micorc*:

```
# crea il file .micorc (da fare solo una volta)
echo –ORBIfaceRepoAddr inet:zirkon:9000 > .micorc
echo –ORBImplRepoAddr inet:zirkon:9001 >> .micorc
```

mandiamo in esecuzione i programmi **ird** e **micod**:

```
# run ird
ird –ORBIIOPAddr inet:zirkon:9000

# run micod
micod –ORBIIOPAddr inet:zirkon:9001
```

Adesso siamo pronti per registrare il server (supponiamo che si chiami molto originalmente *server*). Ricordiamo che implementa l'interfaccia *Account* il cui Repository ID è *IDL:Account:1.0*. Creiamo una voce per lui con:

```
imr create Account shared ~/server IDL:Account:1.0
```

In questo caso abbiamo supposto di trovarci su una macchina Unix e che il programma server si trovi nella home directory. Se fossimo su una macchina con Windows 95/98/NT avremmo dovuto specificare al posto di *~/server* qualcosa del genere *c:\server*. Tutti i percorsi devono essere indicati in modo assoluto.

3.9.1 Gli shared server

In una politica di attivazione **shared server**, più oggetti possono risiedere nello stesso programma o processo. Il BOA attiva il server la prima volta che viene invocata una richiesta su un oggetto implementato da quel server. Una volta che è stato attivato ed inizializzato, il server notificherà al BOA che è pronto a soddisfare le richieste dei client invocando il metodo **impl_is_ready**. Tutte le richieste successive saranno inviate a questo server; il BOA non attiverà nessun altro server per questa implementazione.

Quando il processo è pronto per terminare, avviserà il BOA con la chiamata **deactivate_impl**. A questo punto, tutti gli oggetti che sono in esecuzione all'interno del processo saranno automaticamente disattivati. Si può anche disattivare un singolo oggetto in un qualsiasi momento con la funzione **deactivate_obj**. Potrebbe presentarsi questa necessità quando non ci sono più client attivi che hanno un riferimento a questo oggetto.

È da notare che CORBA non richiede che gli oggetti **shared server** invochino **obj_is_ready** quando vengono attivati per la prima volta, ma comunque è necessario farlo per molte implementazioni dell'ORB. Inoltre, CORBA menziona che gli oggetti possono essere istanziati su richiesta quando si riceve un invocazione, ma lascia la più completa libertà su come implementare questo concetto.

Se si ha la necessità di eseguire più oggetti concorrentemente all'interno dello stesso processo si possono usare i **thread**. In questo caso, entriamo in un area che è fortemente dipendente dall'implementazione dell'ORB; ad esempio Mico, almeno fino alla versione 2.2.7, non supporta affatto i **thread**.

3.9.2 Gli unshared server

Nella politica di attivazione **unshared server**, ogni oggetto risiede in un processo server differente. Un nuovo server è attivato la prima volta che una richiesta è invocata sull'oggetto. Quando l'oggetto ha eseguito la sua fase di inizializzazione, notifica al

BOA che è pronto a gestire le richieste chiamando la funzione **obj_is_ready**. Viene fatto partire un nuovo server ogni volta che si richiede un oggetto che non è già attivo, anche se un server per un altro oggetto con la stessa implementazione è in esecuzione. Un oggetto server rimane attivo e riceverà le richieste dei client finché non invocherà il metodo **deactivate_obj**.

L'uso tipico di questo genere di attivazione si ha in quelle situazioni in cui si usano degli oggetti dedicati. Per esempio, si può avere un oggetto dedicato che rappresenta una stampante oppure un robot in una catena di montaggio. Invece di gestire più oggetti, l'implementazione si occupa di un unico oggetto.

3.9.3 I server per method

Quando si usa la politica di attivazione **server per method**, viene sempre fatto partire un nuovo server ogni volta che viene invocata una richiesta. Il server è in esecuzione solamente durante la durata di quella richiesta. In questa situazione capita spesso che ci siano in esecuzione più processi per lo stesso oggetto concorrentemente attivi.

Siccome parte un nuovo server per ogni richiesta, non è necessario per l'implementazione notificare al BOA quando un oggetto è attivo o è disattivato. Il BOA fa partire un nuovo processo per ogni richiesta, indipendentemente dal fatto che ci sia o meno un'altra richiesta per lo stesso oggetto già in esecuzione.

Probabilmente questa è la politica di attivazione che viene usata di meno rispetto alle altre. Il suo uso tipico potrebbe essere per eseguire degli **script**, oppure per eseguire dei programmi di utilità che vengono mandati in esecuzione di rado.

3.9.4 I persistent server

Nella politica di attivazione **persistent server**, i server sono attivati all'esterno del BOA. Tipicamente è l'utente che li attiva e successivamente avvisa il BOA che sono pronti con la chiamata **impl_is_ready**. Vengono trattati come se fossero degli **shared server**; infatti un processo riceve richieste per metodi e per oggetti diversi. Se l'implementazione non è pronta quando arriva una richiesta, viene restituito un errore sotto forma di eccezione.

I **persistent server** sono quindi un caso speciale degli **shared server**. La differenza consiste nel fatto che i server non vengono attivati dall'ORB. Generalmente si usano per gestire dei database. Un programma CORBA che parte da una riga di comando è un **persistent server**.

3.9.5 I library server

In tutte le politiche di attivazione viste finora, abbiamo assunto che il client e il server siano programmi differenti che sono in esecuzione su processi differenti. Questo approccio ha il vantaggio che le due parti si possono legare fra di loro a tempo di esecuzione, ma ha lo svantaggio che si devono attraversare i confini dei processi con il meccanismo delle chiamate remote.

Il modo di attivazione **library** elimina questo problema e conserva il collegamento dinamico. Infatti, l'implementazione dell'oggetto, in questo caso chiamata *modulo*, viene caricata nello spazio di memoria del client. L'invocazione di un metodo caricato in questo modo è veloce come una semplice chiamata di funzione C++.

Un client, che vuole usare un **server library**, non differisce dagli altri normali client. È il modulo che richiede del codice speciale e una registrazione particolare nell'Implementation Repository.

3.10 Le invocazioni statiche e dinamiche

La Figura 3.5 mostra i due tipi di invocazioni che sono supportate da un ORB CORBA: quelle statiche e quelle dinamiche. In entrambi i casi, il client esegue una richiesta usando un Object Reference e invocando il metodo che offre quel particolare servizio. Il server non si accorge se la chiamata è statica oppure dinamica.

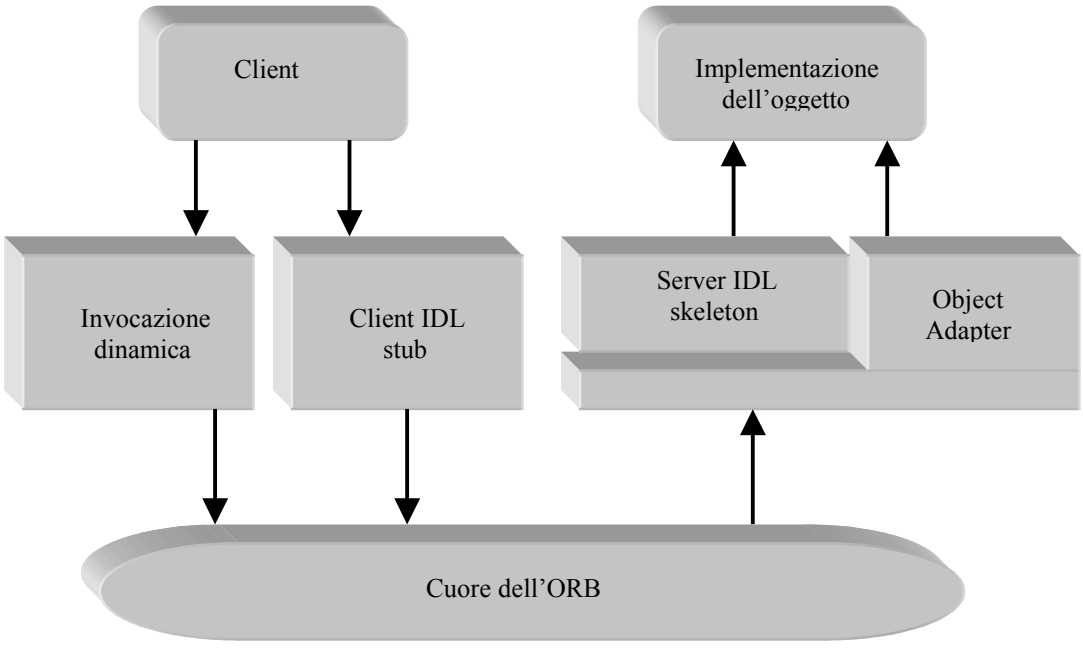


Figura 3.5 Invocazione statica e dinamica dei metodi CORBA.

I client vedono le interfacce degli oggetti attraverso la prospettiva del linguaggio *target* che porta l'ORB al livello del programmatore. I programmi client sono in grado di lavorare senza nessuna modifica al sorgente su qualsiasi ORB che supporta quel determinato linguaggio. Sono in grado di chiamare qualsiasi istanza di un oggetto che implementa l'interfaccia. L'implementazione dell'oggetto, il suo Object Adapter e l'ORB usato per accedervi, sono totalmente trasparenti sia nelle chiamate statiche che in quelle dinamiche.

L'interfaccia statica è direttamente generata in forma di file **stub** dal compilatore IDL. È perfetta per i programmi che conoscono a tempo di compilazione i particolari delle operazioni che vogliono invocare. L'interfaccia **stub** statica è legata a tempo di compilazione e fornisce i seguenti vantaggi rispetto all'invocazione dinamica:

- **È facile da programmare.** I metodi remoti sono chiamati semplicemente invocandoli per mezzo del loro nome e passandogli i parametri. È una forma di programmazione veramente naturale.
- **Fornisce un solido controllo dei tipi.** Il controllo dei tipi è fatto direttamente dal compilatore.
- **Ha delle ottime prestazione.** In alcuni test è stato provato che l'invocazione statica dei metodi è anche 40 volte più veloce della corrispondente invocazione dinamica. La maggior parte del tempo in quest'ultima si perde per la preparazione della chiamata.
- **È autodocumentante.** Si può capire cosa si sta facenda semplicemente leggendo il sorgente.

Al contrario, l'invocazione dinamica dei metodi fornisce un ambiente molto più flessibile. Permette di aggiungere nuove classi al sistema senza cambiare il codice del client. È molto utile per quei **tool** che scoprono quali servizi vengono forniti a tempo di esecuzione. Si può scrivere del codice veramente generico con le API dinamiche. Comunque, la maggior parte delle applicazioni non hanno bisogno di tutta questa flessibilità e lavorano solamente con le invocazioni statiche.

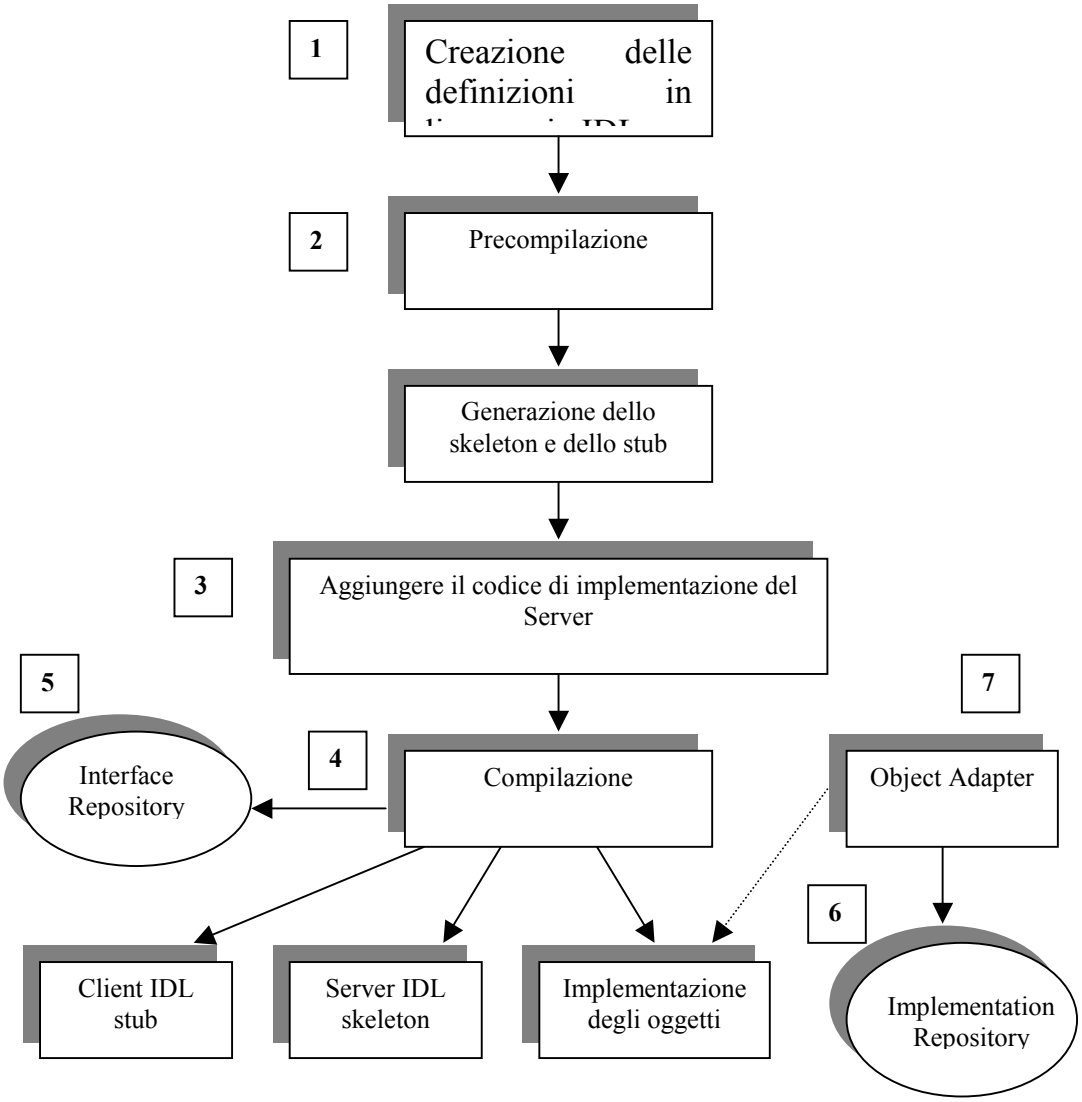
3.10.1 Le invocazioni statiche

Per i programmatori che hanno una certa familiarità con l'architettura client/server le invocazioni statiche possono sembrare molto simili alle **RPC**. Invece, per quelli che provengono dal mondo Java, C++, Smalltalk, le invocazioni statiche sono come un'ordinaria chiamata di funzione, eccetto per il fatto che è remota. Idealmente, non si dovrebbe essere in grado di dire se una chiamata è locale oppure remota. Spetta all'ORB di preoccuparsi di tutti i dettagli. In pratica, però, una certa differenza semantica continua ad esistere. Per esempio, si deve ottenere un riferimento ad un oggetto prima di poter invocare un qualsiasi suo metodo. Comunque, sono proprio gli

Object Reference a rendere CORBA molto più potente delle altre architetture client/server presenti attualmente sul mercato.

Un Object Reference CORBA è un mezzo molto potente per le architetture distribuite. Esso punta all'interfaccia di un oggetto, cioè un insieme di metodi in relazione che operano su un singolo oggetto. Al contrario, tanto per fare un esempio, le RPC restituiscono solamente un riferimento ad una singola funzione. Inoltre, le interfacce CORBA si possono comporre tramite l'ereditarietà multipla e gli oggetti sono polimorfici, nel senso che la stessa chiamata può avere effetti differenti in base all'oggetto che l'ha ricevuta.

La Figura 3.6 mostra i passi che si devono seguire per creare le proprie classi server, fornire le interfacce **stub** per esse, memorizzare la loro definizione nell'Interface Repository, istanziare gli oggetti a tempo di esecuzione e registrare la loro presenza nell'Implementation Repository.



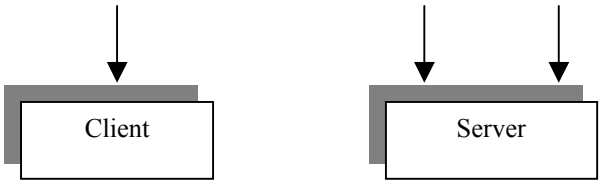


Figura 3.6 Processo di creazione di un'interfaccia statica tra client e server.

Diamo adesso una spiegazione dei passi che si devono eseguire:

1. **Definizione delle proprie classi di oggetti usando l'Interface Definition Language (IDL).** L'IDL è il mezzo grazie al quale gli oggetti informano i loro potenziali clienti di quali operazioni sono disponibili e di come devono essere invocate. Questo linguaggio definisce il tipo degli oggetti, i loro attributi, i metodi che esportano e i loro parametri.
2. **Precompilare il file IDL.** Un tipico preprocessore CORBA analizza il file IDL e produce lo stub del client e lo skeleton del server nel linguaggio target. Questi file saranno usati nelle implementazioni fornite dal programmatore.
3. **Aggiungere il codice di implementazione allo skeleton.** Spetta ovviamente allo sviluppatore del componente aggiungere il codice che implementa i metodi allo skeleton. In altre parole, deve creare le classi server.
4. **Compilazione del codice.** Un compilatore CORBA tipico è capace di generare almeno tre tipi di file: 1) gli *import file* che descrivono gli oggetti per un'Interface Repository; 2) gli *stub del client* per i metodi definiti in IDL (saranno poi usati dai programmi client quando invocheranno dei metodi tramite l'interfaccia statica); 3) gli *skeleton del server* che chiamano i metodi dell'implementazione. Spetta al programmatore fornire il codice che implementa le classi server. La generazione automatica di questi file libera il programmatore dall'onere di scriverli e libera le applicazioni dalle dipendenze dalla particolare implementazione dell'ORB. Ovviamente il tutto deve essere compilato insieme per ottenere da una parte il server e dall'altra il client.
5. **Legare la definizione delle classi all'Interface Repository.** Tipicamente si usa un'*utility*, fornita dalla distribuzione CORBA, per questo processo. Comunque, questo è un passo che si esegue sempre quando si sviluppano dei server, ma che serve solo per le invocazioni dinamiche.
6. **Registrare gli oggetti a tempo di esecuzione nell'Implementation Repository.** L'Object Adapter registra nell'Implementation Repository i riferimenti agli oggetti e il tipo di ogni oggetto istanziato dal server. L'Implementation Repository conosce anche quali classi di oggetti sono supportate da un particolare server. L'ORB usa queste informazioni per localizzare gli oggetti attivi o per richiedere l'attivazione di un oggetto su un particolare server.
7. **Istanziare gli oggetti sul server.** In fase di inizializzazione, l'Object Adapter crea gli oggetti server che forniscono i servizi remoti richiesti dai client. Questi oggetti a tempo di esecuzione sono istanze delle classi del server. Abbiamo visto che

CORBA specifica differenti strategie che sono usate dall'Object Adapter per creare e gestire gli oggetti a tempo di esecuzione.

In conclusione, l'invocazione statica dei metodi è realmente molto semplice. La maggior parte del lavoro è fatta dall'ORB. L'unico lavoro in più che deve fare il programmatore rispetto alla programmazione tradizionale è quello di scrivere le definizioni delle interfacce degli oggetti in IDL. Comunque, anche questo passo può essere semplificato dato che ci sono dei programmi, come ad esempio il *Caffeine* di Netscape/Visigen che genera automaticamente lo stub e lo skeleton processando direttamente il sorgente scritto col linguaggio *target*.

L'ultima cosa da notare nelle chiamate statiche è la gestione degli stub. Come visto in precedenza, il client deve essere in possesso sia di un riferimento ad un oggetto, sia dello stub prima di poter invocare un metoto remoto sul server. Se è vero che è facile passare un object reference, non vale la stessa cosa per lo stub. Infatti, deve far parte del codice del client stesso. Questo è il principale, e forse unico, svantaggio delle invocazioni statiche. Se il linguaggio *target* è Java, gli stub si potebbero scaricare in forma di *bytecode*, e si potrebbero successivamente usare per collegarsi staticamente agli oggetti server.

Facciamo un esempio completo di collegamento statico. Ripercorriamo i passi visti in precedenza, ma questa volta forniremo le operazioni da fare specifiche di Mico.

Passo 1. Definizione delle proprie classi di oggetti usando l'IDL.

Definiamo un oggetto *Account* in questo modo:

```
interface Account
{
   void deposit (in unsigned long amount);
   void withdraw (in unsigned long amount);
   long balance();
};
```

Salviamo questa definizione nel file *account.idl*.

Passo 2. Precompilare il file IDL.

Il compilatore IDL di Mico si chiama idl e si usa in questo modo:

```
idl account.idl
```

Otterremo come risultato due file: *account.h* e *account.cc*. Il primo contiene la dichiarazione delle classi di base che verranno ereditate dall'implementazione dell'oggetto, del codice **stub** che serve al client e del codice **skeleton** usato dal server. Il secondo contiene la loro implementazione e del codice di supporto.

Passo 3. Aggiungere il codice di implementazione allo skeleton.

Per ogni interfaccia dichiarata nel file IDL, il compilatore generera tre classi C++. Quindi, nel nostro caso avremo le classi: *Account*, *Account_skel* e *Account_stub*.

La prima serve da classe di base. Essa contiene tutte le definizioni che appartengono all'interfaccia *Account*, le dichiarazioni locali e le strutture di dati definite dall'utente. Definisce anche una funzione virtuale per ogni metodo contenuto nell'interfaccia. Ecco un suo piccolo frammento:

```
// Codice estratto dal file account.h
class Account : virtual public CORBA::Object
{
   ...
public:
   ...
   virtual void deposit (CORBA::ULong amount) = 0;
   virtual void withdraw (CORBA::ULong amount) = 0;
   virtual CORBA::Long balance () = 0;
};
```

La classe *Account_skel* è derivata da *Account*. Essa aggiunge il codice per le chiamate remote, ma non definisce le funzioni virtuali. Quindi rimane una classe astratta che non può essere istanziata. Per implementare l'oggetto **Account** si deve creare una classe che eredita da *Account_skel* e si deve fornire il codice per le funzioni **deposit**, **withdraw** e **balance**. Ecco come fare:

```
#include "account.h"

class Account_impl : virtual public Account_skel
{
private:
   CORBA::Long _current_balance;

public:
   Account_impl()
   {
      _current_balance = 0;
   };
   void deposit(CORBA::ULong amount)
   {
      _current_balance += amount;
   };
   void withdraw(CORBA::ULong amount)
   {
      _current_balance -= amount;
   };
   CORBA::Long balance()
   {
      return _current_balance;
   };
```

```
};
```

La classe *Account_stub* è anche essa derivata dalla classe *Account*, ma non è una classe astratta perché definisce le funzioni virtuali pure. L'implementazione di queste funzioni è fornita automaticamente dal compilatore IDL e si occupa del **marshalling** dei parametri. Vediamo anche qui un suo frammento:

```
// Codice estratto dal file account.h
class Account;
typedef Account *Account_ptr;

class Account_stub : virtual public Account
{
   ...
public:
   ...
   void deposit(CORBA::ULong amount)
   {
      // Codice di marshalling per deposit
   }
   void withdraw(CORBA::ULong amount)
   {
      // Codice di marshalling per withdraw
   }
   CORBA::Long balance()
   {
      // Codice di marshalling per balance
   }
};
```

Ciò rende *Account_stub* una classe concreta C++ che può essere istanziata. Il programmatore, comunque, non la userà mai direttamente, ma si limiterà ad usare la classe *Account*.

Passo 4. Compilazione del codice.

Prima di compilare il tutto, scriviamo il codice che completa sia il server che il client. Creiamo un nuovo file, che chiameremo *account_server.cc*, e inseriamoci la definizione della classe *Account_impl* vista in precedenza e il codice che completa il server:

```
// file account_server

#include "account.h"

class Account_impl : virtual public Account_skel
{
   ...
};

int main (int argc, char *argv[])
```

```
{
   // Inizializzazione dell'ORB
   CORBA::ORB_var orb =
      CORBA::ORB_init(argc, argv, "mico-local-orb");
   CORBA::BOA_var boa =
      orb->BOA_init(argc, argv, "mico-local-boa");

   Account_impl *server = new Account_impl;

   boa->impl_is_ready(CORBA::ImplementationDef::_nil());
   orb->run();
   CORBA::release(server);

   return 0;
};
```

L'oggetto viene creato con una semplice chiamata a *new* del C++. Con il metodo **impl_is_ready** avvisiamo il BOA che l'oggetto è stato creato. Abbiamo supposto che il server sia di tipo **shared**, altrimenti avremmo dovuto fare una chiamata diversa, come ad esempio **obj_is_ready** nel caso di server **unshared**. Normalmente si dovrebbe cercare all'interno dell'Implementation Repository una voce relativa al server e passarla come parametro al metodo **impl_is_ready**, ma in questo caso abbiamo passato un puntatore nullo (*CORBA::ImplementationDef::_nil*) e quindi sarà il BOA ad occuparsi del tutto. Il metodo dell'ORB **run** entra in un ciclo che processerà le eventuali invocazioni. Lo standard CORBA prevede che tutti gli oggetti devono essere distrutti con una chiamata al metodo **release**. Se si usano gli **smart pointer** (le variabili che terminano con *_var*), il metodo **release** viene chiamato automaticamente. In questo caso, siccome abbiamo usato un puntatore normale, dobbiamo fare una chiamata esplicita.

Passiamo al codice del client. Creiamo un nuovo file, che chiameremo *account_client.cc*, e inseriamoci questo codice:

```
// file account_client.cc

#include "account.h"

int main (int argc, char *argv[])
{
   // Inizializzazione dell'ORB
   CORBA::ORB_var orb =
      CORBA::ORB_init(argc, argv, "mico-local-orb");
   CORBA::BOA_var boa =
      orb->BOA_init(argc, argv, "mico-local-boa");

   CORBA::Object_var obj =
      orb->bind("IDL:Account:1.0");
   if (CORBA::is_nil(obj)) {
```

```
        // Collegamento fallito
    }
    Account_var client = Account::_narrow(obj);

    client->deposit(700);
    client->withdraw(250);
    cout << "Bilancio : " << client->balance() << endl;

    return 0;
};
```

Dopo aver inizializzato l'ORB e il BOA, il client usa il metodo **bind** per legarsi all'oggetto che ha come Repository ID *IDL:Account:1.0* (vedremo ai passi 5 e 6 come associare l'identificatore all'oggetto). È da notare che in questo caso non abbiamo specificato nessun indirizzo particolare nel codice del **bind**, ma manderemo in esecuzione il client con l'opzione da riga di comando **–ORBBindAddr** per indicargli gli indirizzi dove cercare.

Passiamo alla compilazione. È complicato compilare e *linkare* le applicazioni Mico perché si devono indicare i flag specifici di un sistema e le librerie da usare. Per questo motivo Mico fornisce dei file **script** che vengono creati in fase di configurazione e che semplificano la vita del programmatore:

- **mico-c++.** È il compilatore C++.
- **mico-ld.** È il linker.
- **mico-shc++.** È il compilatore C++ che si deve usare quando si creano dei moduli caricabili dinamicamente, come ad esempio i server **library**.
- **mico-shld.** È il linker che si deve usare quando si creano i moduli come al punto precedente.

Questi **script** si riferiscono ad un ambiente Unix compatibile, ma vengono creati anche su Windows 95/98/NT se il compilatore C++ istallato è il **Cygnus CDK**.

Per compilare il nostro esempio dobbiamo dare i seguenti comandi:

```
mico-c++ -I. –c account.cc –o account.o
mico-c++ -I. –c account_server.cc –o account_server.o
mico-c++ -I. –c account_client.cc –o account_client.o
mico-ld –o server account.o account_server.o –lmico2.2.7
mico-ld –o client account.o account_client.o –lmico2.2.7
```

Passo 5. Legare la definizione delle classi all'Interface Repository.

Fortunatamente di questo compito si occupa il compilatore IDL. Infatti, con l'opzione **--emit-repoids** indichiamo al compilatore di generare automaticamente le istruzioni *#pragma* per i Repository ID, mentre con l'opzione **--feed-ir** si inseriscono le definizioni nell'Interface Repository. I

Repository ID possono essere anche essere inseriti manualmente dall'utente nel file IDL.

Prima di tutto facciamo generare i Repository ID automaticamente:

```
idl --emit-repoids --codegen-idl --name=account1.idl \
   --no-codegen-c++ account.idl
```

Questo comando genererà il file *account1.idl* che sarà identico ad *account.idl*, ma conterrà al suo interno anche i Repository ID. Ecco il suo contenuto:

```
// file account1.idl

#pragma ID Account "IDL:Account:1.0"
   interface Account
   {
      void deposit (in unsigned long amount);
      void withdraw (in unsigned long amount);
      long balance();
   };
```

Ovviamente l'istruzione #pragma poteva essere inserita manualmente dal programmatore, ad esempio per avere un numero di versione differente.

Avviamo l'Interface Repository assegnandole un indirizzo:

```
ird -ORBIIOPAddr inet:localhost:9000
```

e inseriamo al suo interno le definizioni del file *account1.idl*:

```
idl --feed-ir account1.idl -ORBIfaceRepoAddr \
   inet:localhost:9000
```

Questo passo non è necessario se si usa solamente l'invocazione statica dei metodi, ma è un passo che eseguiamo per completezza.

Passo 6. Registrare gli oggetti nell'Implementation Repository.

Avviamo il **BOA daemon**:

```
micod -ORBIIOPAddr inet:localhost:9001
```

e registriamo il server:

```
imr create Account shared ~/server IDL:Account:1.0 \
   -ORBImplRepoAddr inet:localhost:9001
```

Passo 7. Istanziare gli oggetti sul server.

Le istanze degli oggetti vengono create dal BOA quando viene invocato un loro servizio. In realtà il BOA fa partire il server ed è quest'ultimo che crea le istanze degli oggetti.

L'ultima cosa da fare è avviare il client indicandogli l'indirizzo, del **BOA daemon**, che verrà usato nella chiamata a **bind**:

```
client -ORBBindAddr inet:localhost:9001
```

3.10.2 Le invocazioni dinamiche

Vediamo adesso il collegamento dinamico, cioè la **Dynamic Invocation Interface** (**DII**). Come abbiamo detto in precedenza, questo approccio non ha bisogno di nessuno stub, ma il riferimento all'oggetto è comunque necessario. La DII permette ai client di scegliere un qualsiasi oggetto a tempo di esecuzione e poi di invocare dinamicamente i suoi metodi. I client possono invocare qualsiasi operazione di qualsiasi oggetto senza aver bisogno di uno stub precompilato. Questo significa che i client scoprono le informazioni relative ad una interfaccia nel momento in cui devono fare un'invocazione, non è richiesta nessuna conoscenza a tempo di compilazione.

Nelle architetture client/server, la DII è quella che sicuramente offre la maggiore libertà di azione. I server offrono i nuovi servizi e le nuove interfacce non appena diventano disponibili. I client scoprono queste interfacce e come utilizzarle a tempo di esecuzione. La DII fornisce un ambiente veramente dinamico che permette al sistema di rimanere flessibile e estensibile, una qualità veramente apprezzabile soprattutto nel mondo di Internet.

I client possono scoprire le caratteristiche degli oggetti server in diversi modi. In quello più semplice basta passargli un object reference in formato stringa. Il client provvederà a riconvertire la stringa e a stabilire la connessione. I client possono anche cercare gli oggetti per nome usando il **Naming Service** di CORBA (di cui daremo una breve descrizione successivamente), oppure scoprirli tramite una specie di elenco che è chiamato **Trader Service**.

Prima di poter invocare dinamicamente un metodo su un oggetto, è necessario cercare l'oggetto stesso e ottenere un suo riferimento. Una volta fatto ciò, si userà il riferimento per ottenere l'interfaccia e per costruire dinamicamente la richiesta. Si deve specificare nella richiesta il metodo che si vuole eseguire e i suoi parametri. Tipicamente, queste informazioni si ottengono dall'Interface Repository, ma si può usare anche il **Trader Service** che fornisce delle informazioni più approfondite come ad esempio l'intervallo dei valori che un server si aspetta.

Supponiamo di aver ottenuto, in qualche modo, un riferimento all'oggetto che si vuole invocare dinamicamente. Questa è la descrizione ad alto livello di cosa si deve fare per chiamare un metodo remoto:

1. **Ottenere il nome dell'interfaccia.** Gli oggetti CORBA sono introspettivi, cioè ci possono fornire molte informazioni su loro stessi. Conseguentemente, possiamo chiedere all'oggetto il nome della sua interfaccia invocando il metodo **get_interface**. Questa chiamata restituisce un riferimento ad un oggetto **InterfaceDef** che descrive l'interfaccia e che si trova nell'Interface Repository.
2. **Ottenere una descrizione del metodo dall'Interface Repository.** Possiamo usare l'InterfaceDef come punto di entrata per *navigare* all'interno dell'Interface Repository. È possibile ottenere ogni tipo di informazione attinente all'interfaccia e ai metodi che supporta. CORBA specifica circa 10 metodi per cercare all'interno dell'Interface Repository e per descrivere gli oggetti che contiene. Nel nostro caso il client si servirà della funzione **lookup_name** per cercare il metodo che vuole invocare e successivamente della chiamata **describe** per ottenere la definizione completa in IDL del metodo. In alternativa si può usare la funzione **describe_interface** per ottenere una descrizione di tutta l'interfaccia e cercare il metodo che si vuole invocare.
3. **Creare una lista di argomenti.** CORBA specifica una struttura dati per passare i parametri, che è chiamata **Named Value List**. Si implementerà questa lista usando lo pseudo oggetto **NVList**. La lista può essere creata usando la funzione **create_list** e le varie voci si aggiungono con la funzione **add_item**. In alternativa, si può ricorrere all'aiuto dell'ORB per creare la lista chiamando la funzione **create_operation_list** dell'oggetto CORBA::ORB. Basta passargli il nome dell'operazione per la quale si vuole ricevere la lista.
4. **Creare la richiesta.** Una richiesta è uno pseudo oggetto CORBA che contiene il nome del metodo, la lista degli argomenti e il valore di ritorno. Si può creare la richiesta invocando **create_request**. I parametri che accetta sono: il nome del metodo da chiamare, lo pseudo oggetto **NVList** e un puntatore al valore di ritorno. In alternativa, si può creare una versione breve della richiesta usando la funzione **_request** e passandogli solamente il nome del metodo da invocare, i parametri non sono necessari.
5. **Invocare la richiesta.** Si può invocare la richiesta in uno di questi tre modi: 1) usando la chiamata **invoke** per inviare la richiesta e ottenere il risultato; 2) usando **send_deffered** per inviare la richiesta e per restituire immediatamente il controllo al programma (si dovrà successivamente usare una di queste due funzioni per ottenere il risultato: **poll_response** o **get_response**); 3) usando **send_oneway** che non presuppone nessun risultato o risposta. Questi tre stili di invocazioni sono indicati rispettivamente con i nomi **sincrono**, **asincrono** e **oneway**.

Come si può vedere l'invocazione dinamica di un metodo richiede qualche sforzo in più rispetto all'invocazione statica. La maggior parte del lavoro è dovuta alla creazione della richiesta. Questo processo può essere reso ancora più complicato dai differenti modi in cui si può costruire e invocare una richiesta.

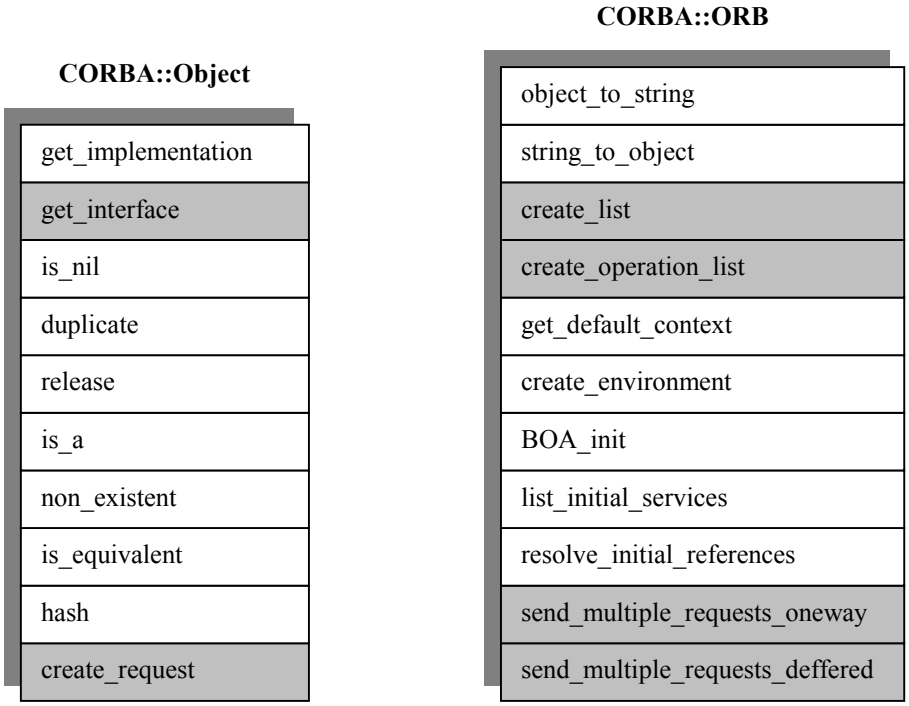
3.10.2.1 L'interfaccia dell'invocazione dinamica

I servizi di cui si ha bisogno per invocare dinamicamente un oggetto fanno parte del nucleo di CORBA. Sfortunatamente, questi metodi sono sparsi su quattro interfacce che fanno parte del modulo CORBA. La Figura 3.7 mostra le quattro interfacce con tutti i loro metodi, ma solo quelli evidenziati in grigio si riferiscono alle invocazioni dinamiche.

Queste interfacce sono implementate nella forma di **pseudo oggetti**. Uno pseudo oggetto è un oggetto che da parte della distribuzione CORBA e che l'ORB crea direttamente, ma che può essere invocato come un qualsiasi altro oggetto. L'ORB stesso è uno pseudo oggetto. Si possono invocare i metodi degli pseudo oggetti dell'ORB sia dal lato client che dal lato server.

Vediamo adesso una breve descrizione dei metodi di queste quattro interfacce che si riferiscono alle invocazioni dinamiche:

- **CORBA::Object** è l'interfaccia di uno pseudo oggetto che definisce le operazioni che ogni oggetto CORBA deve supportare (queste operazioni sono fornite dall'ORB, basta semplicemente ereditarle quando si definiscono i propri oggetti). È l'interfaccia radice per tutti gli oggetti CORBA. Questa interfaccia include tre metodi che si devono utilizzare per costruire le invocazioni dinamiche. Si deve usare la funzione **get_interface** per ottenere l'interfaccia che un oggetto supporta. La chiamata restituisce un riferimento ad un oggetto **InterfaceDef** che è un oggetto, all'interno dell'Interface Repository, che descrive l'interfaccia. Con la chiamata **create_request** si crea un oggetto **Request**, al quale si deve passare il nome del metodo e i parametri richiesti per l'invocazione remota dei metodi. In alternativa, si può creare una versione breve della richiesta invocando la funzione **_request** e passargli solamente il nome del metodo.



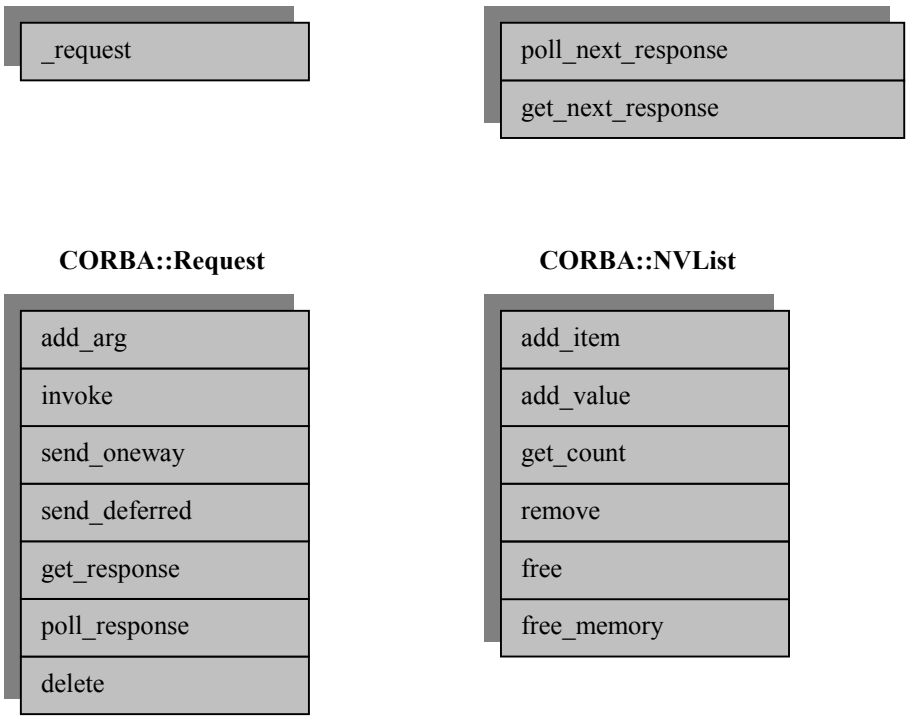


Figura 3.7 Le interfacce per l'invocazione dinamica dei metodi.

- **CORBA::Request** è l'interfaccia di uno pseudo oggetto che definisce le operazioni su un oggetto remoto. Il metodo **add_arg** aggiunge incrementalmente gli argomenti alla richiesta. Il metodo **invoke** esegue la chiamata ed attende il risultato. Il metodo **send_deferred** invia la richiesta e restituisce subito il controllo al chiamante senza attendere il risultato. I messaggi di ritorno possono essere successivamente intercettati usando la funzione **poll_response** e possono essere letti con **get_response**. Se non si ha bisogno di nessuna risposta si può usare la chiamata **send_oneway** che invia la richiesta e restituisce subito il controllo senza attendere il risultato. Il metodo **delete** serve per rimuovere l'oggetto **Request** dalla memoria.
- **CORBA::NVList** è l'interfaccia di uno pseudo oggetto che aiuta il programmatore nella costruzione della lista dei parametri. Un oggetto **NVList** mantiene una lista di dati autodescriventi chiamati *NamedValue*. L'interfaccia NVList definisce le operazioni che permettono di manipolare la lista. Si deve usare il metodo **add_item** per aggiungere un parametro alla lista. Si può settare il suo valore con la funzione **add_value**. Con la chiamata **get_count** si ottiene il numero totale degli elementi allocati nella lista. Si può rimuovere una voce usando la funzione **remove**. Il metodo **free_memory** serve per liberare la memoria allocata dinamicamente per contenere gli argomenti della lista, mentre il metodo **free** libera la memoria che contiene la lista stessa. Quest'ultimo chiama al suo interno **free_memory** per ogni elemento della lista.

- **CORBA::ORB** è l'interfaccia di uno pseudo oggetto che definisce i metodi di uso generico dell'ORB. Si possono invocare questi metodi sia da parte del client che da parte del server. Sei di questi metodi sono specifici per la costruzione di una richiesta dinamica. Il metodo **create_list** serve per creare un oggetto NVList vuoto che si deve successivamente riempire. In alternativa, si può usare il metodo **create_operation_list** per far fare la maggior parte del lavoro all'ORB. Questo metodo crea l'oggetto NVList e automaticamente lo riempie con la descrizione degli argomenti necessari per l'operazione remota che si desidera invocare. L'ORB fornisce quattro metodi per inviare e ricevere delle richieste multiple. Con **send_multiple_requests_oneway** si inviano delle richieste che non necessitano di nessuna risposta. **Send_multiple_requests_deferred** si usa, invece, per inviare le richieste multiple in modo asincrono. Le risposte vengono successivamente intercettate e lette rispettivamente con i metodi **poll_next_response** e **get_next_response**.

Oltre a queste quattro interfacce, si può usare un oggetto Interface Repository per costruire un'invocazione remota, ma è un processo molto più complicato.

3.10.3 I test comparativi

Vediamo adesso alcuni test che confrontano le prestazioni delle invocazioni statiche e di quelle dinamiche. I dati si riferiscono ad alcune prove effettuate, con l'ORB **VisiBroker for Java** su una LAN Ethernet, dagli autori del libro *"Client/Server Programming with Java and CORBA"*.

I test sono stati effettuati sia con l'*overhead* associato alla creazione della richiesta, sia senza. La Figura 3.8 mostra i risultati.

Invocazione statica	**Invocazione dinamica**	
3.2 msec	**3.3 msec**	**131.4 msec**

Figura 3.8 Confronto delle prestazioni tra invocazione statica e invocazione dinamica dei metodi. I tempi si riferiscono alla media dei tempi di risposta di 1000 chiamate di funzioni remote.

Come si può vedere, i numeri mostrano che le invocazioni dinamiche sono circa 40 volte più lente delle corrispondenti chiamate statiche. La sorpresa è che la maggior parte del tempo si perde nella preparazione della richiesta. Non è molto complicato capirne il motivo. Infatti, la preparazione della richiesta presuppone molti accessi all'Interface Repository che fanno perdere molti cicli di clock e che implicano anche degli accessi al disco.

La programmazione tradizionale CORBA fornisce, quindi, sia un motodo statico per l'invocazione dei metodi, sia uno dinamico. Se il linguaggio *target* è Java si ha a disposizione una terza scelta: gli stub scaricabili. In questo caso sono le **applet** che scelgono gli oggetti che desiderano utilizzare e scaricano dalla rete i loro stub.

La scelta di quale metodo utilizzare dipende dall'uso che si deve fare degli oggetti remoti. Se ad esempio i client invocano gli oggetti server frequentemente e questi non cambiano, conviene usare l'approccio statico. Se invece gli oggetti vengono invocati di rado la soluzione migliore è quella dinamica che non appesantisce il client con gli stub. Con il linguaggio Java, se i client *girano* all'interno di **browser**, si possono usare delle **applet** che scoprono a tempo di esecuzione gli oggetti esistenti e scaricano i loro stub per effettuare il collegamento.

Capitolo 4

Architettura dell'integrazione

In questo capitolo descriveremo la specifica dell'**OMG group** per la comunicazione tra CORBA e COM.

Una specifica ottimale dovrebbe permettere agli oggetti, di entrambi i sistemi, di rendere le loro funzionalità chiave visibili ai client dell'altro sistema, nel modo più trasparente possibile. La specifica per l'architettura dell'integrazione è stata creata per raggiungere questo scopo.

4.1 Scopo dell'architettura d'integrazione

Lo scopo di quest'architettura è quello di specificare un supporto per una comunicazione bidirezionale tra oggetti COM e oggetti CORBA. Il fine è di avere un oggetto di un modello che è in grado di essere visto come se fosse dell'altro modello. Per esempio, un client che lavora nel modello CORBA deve essere in grado di vedere un oggetto COM come se fosse un oggetto CORBA. Allo stesso modo, un client che lavora nel modello COM deve essere in grado di vedere un oggetto CORBA come se fosse un oggetto COM.

Ci sono molte cose simili nei due sistemi. In particolare, entrambi sono centrati attorno all'idea che un oggetto è un'unità discreta di funzionalità che presenta il suo comportamento attraverso un insieme di interfacce completamente descritte. Ogni sistema nasconde i dettagli dell'implementazione ai suoi client. Ad una visione sommaria, COM e CORBA appaiono semanticamente isomorfi.

La maggior parte della specifica dell'integrazione dei due sistemi comporta semplicemente una *mappatura* della sintassi, della struttura e delle possibilità da un sistema all'altro.

Ci sono, comunque, notevoli differenze nei due modelli. COM e CORBA hanno un modo diverso per descrivere cosa è un oggetto, come è tipicamente usato e come i

componenti del modello sono organizzati. Tutte queste cose fanno nascere molti problemi quando si cerca di rendere trasparente la *mappatura*.

4.1.1 Confronto tra oggetti COM e oggetti CORBA

Da un punto di vista COM, un oggetto è tipicamente un sottocomponente di un'applicazione e rappresenta un punto di esposizione alle altre parti dell'applicazione o alle altre applicazioni. Storicamente, il tipico dominio di un oggetto COM è a singolo utente in un ambiente di lavoro Microsoft Windows. Attualmente, lo scopo principale di COM e OLE è di accelerare la collaborazione e la condivisione di informazioni tra applicazioni che usano lo stesso **desktop**, attraverso la manipolazione dell'utente di oggetti visuali (per esempio il **drag and drop**, **cut and paste**, ecc.).

Da un punto di vista CORBA, un oggetto è un componente indipendente che fornisce un insieme di comportamenti in relazione fra di loro. Da un oggetto ci si aspetta che sia trasparentemente disponibile a qualsiasi client CORBA, indipendentemente dalla locazione (o implementazione) degli altri oggetti o client. Molti oggetti CORBA sono centrati su ambienti eterogenei e distribuiti. Storicamente, il tipico dominio di un oggetto CORBA è una rete distribuita arbitrariamente scalabile. Nella sua forma corrente, lo scopo principale di CORBA è di permettere a questi componenti indipendenti di essere condivisi tra una varietà di applicazioni (e altri oggetti), ognuno dei quali può essere eventualmente non in relazione con gli altri.

Naturalmente, CORBA è già usato per definire oggetti **desktop** (vedi ad esempio il **KDE** di Linux) e COM può essere esteso per lavorare in una rete. Allo stesso modo, entrambi i modelli si stanno sviluppando ed evolvendo, e probabilmente diventeranno due sistemi equivalenti in futuro. Quindi, una buona architettura di integrazione deve mappare un sistema sull'altro conservando lo stile che un programmatore si aspetta di trovare.

La cosa più ovvia che i due modelli hanno in comune è l'architettura basata su oggetti (Figura 4.1).

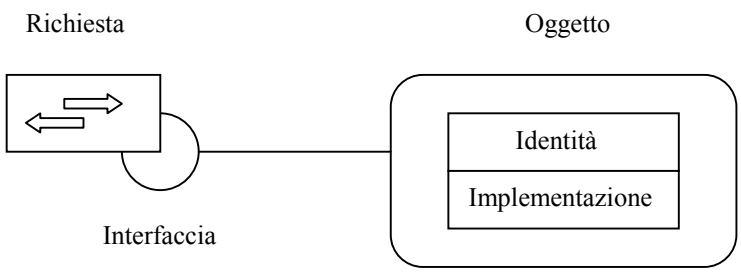


Figura 4.1 Architettura comune degli oggetti COM e CORBA

4.2 Il modello di integrazione

Vediamo adesso le caratteristiche principali del modello CORBA e del modello COM e una breve descrizione di come dovrebbe essere il **modello di integrazione**.

4.2.1 Il modello CORBA

Nel modello CORBA, ogni oggetto è semplicemente un'entità discreta di funzionalità che si presentano attraverso un'interfaccia pubblica. Un'interfaccia (e le funzionalità che stanno alla base) è accessibile attraverso una forma di richiesta. Ogni richiesta è indirizzata ad un oggetto specifico (un'istanza di un oggetto), basata su un riferimento alla sua identità. L'oggetto ricevente deve servire la richiesta invocando il comportamento che ci si aspetta dalla sua specifica implementazione. I parametri della richiesta sono riferimenti ad oggetti oppure valori appartenenti al sistema dei tipi del modello. Le interfacce possono essere composte combinando altre interfacce seguendo le regole di composizione. Le interfacce sono descritte in un linguaggio specifico o possono essere rappresentate sotto forma di librerie.

Il cuore di CORBA è l'**Object Request Broker** o **ORB**. Funzionalmente l'ORB serve per la registrazione delle seguenti cose:

- I tipi e le interfacce, come sono descritti nell'**OMG Interface Definition Language** o **OMG IDL** o **CORBA IDL**.
- L'identità delle istanze, dalle quali l'ORB può costruire dei riferimenti appropriati ad ogni oggetto a cui sono interessati i client.

Un oggetto CORBA può quindi ricevere delle richieste dai client che hanno il suo riferimento e hanno le informazioni necessarie per formulare un'appropriata richiesta all'interfaccia dell'oggetto. Questa richiesta può essere definita staticamente a tempo di compilazione oppure creata dinamicamente a *runtime* in base alle informazioni sui tipi disponibili attraverso un registro dei tipi delle interfacce.

CORBA specifica l'esistenza di un tipo di implementazione chiamata **ImplementationDef** (è un'**Implementation Repository** che contiene la descrizione di questi tipi), ma non specifica come questi due elementi, ImplementationDef e Implementation Repository, devono essere. Così la loro descrizione e il tipo dell'implementazione variano da ORB a ORB (ogni produttore ha la sua implementazione specifica) e non fanno parte della specifica.

4.2.2 Il modello OLE/COM

Il cuore del modello OLE/COM è il **Component Object Module** o appunto **COM**. Funzionalmente, COM permette ad un oggetto di esporre le sue interfacce in una forma binaria (che è una tabella di funzioni virtuali) così ogni client che ha una conoscenza statica a tempo di compilazione della struttura dell'interfaccia e con un riferimento ad un'istanza di un oggetto che espone questa interfaccia, può inviare delle specifiche richieste. Molte interfacce COM sono descritte nel **Microsoft Interface Definition Language** o **MIDL**.

COM supporta un meccanismo di implementazione dei tipi centrato attorno al concetto di classe COM. Una classe COM ha un'identità ben definita e c'è un meccanismo (il registro di Windows) che mappa le implementazioni (identificate dal **CLSID**) con le specifiche unità di codice eseguibile.

COM fornisce anche l'estensione **OLE Automation**. Le interfacce che sono compatibili con OLE Automation devono essere descritte nell'**Object Definition Language** o **ODL** e possono essere opzionalmente registrate in **Type Library** binarie. Le interfacce OLE possono essere invocate dinamicamente da un client che non ha nessuna conoscenza a tempo di compilazione dell'interfaccia, attraverso una speciale interfaccia COM che è **IDispatch**. Si può fare il *type checking* a *runtime* sulle invocazioni quando viene fornita la Type Library. Le interfacce OLE hanno proprietà e metodi, mentre le interfacce COM hanno solo i metodi. I tipi di dato che possono essere usati come parametri nei metodi e nelle proprietà di oggetti OLE sono solo un sottoinsieme dei tipi supportati da COM e non consentono l'uso di tipi definiti dall'utente.

Di conseguenza, l'integrazione con oggetti che espongono interfacce OLE è notevolmente differente da quella con oggetti COM. Sebbene Automation sia implementata attraverso COM, le notevoli differenze fanno in modo che i due modelli siano considerati distinti. Per questo motivo, oltre al *mapping* tra CORBA e COM, vedremo anche quello tra CORBA e OLE.

4.2.3 Le basi del modello di integrazione

Se non si entra nei particolari, Microsoft COM e OMG CORBA sembrano molto simili. Approssimativamente parlando, le interfacce COM (incluse le interfacce Automation) sono equivalenti alle interfacce CORBA. Inoltre, i puntatori alle interfacce COM sembrano equivalenti ai puntatori alle interfacce CORBA. Però, entrando nei particolari (i tipi di dato, la creazione degli oggetti, ecc.) le due architetture sono molto meno isomorfe. Comunque un ragionevole livello di integrazione è possibile forzando il *mapping* in quei casi in cui i due sistemi non sono equivalenti.

Un esempio di come può essere realizzata questa integrazione è illustrato nella Figura 4.2. Essa mostra come un oggetto del sistema B può essere mappato e presentato ad un client del sistema A (chiameremo questa interazione **B/A mapping**). Per esempio, il mapping di un oggetto CORBA per renderlo visibile ad un client COM è un CORBA/COM mapping.

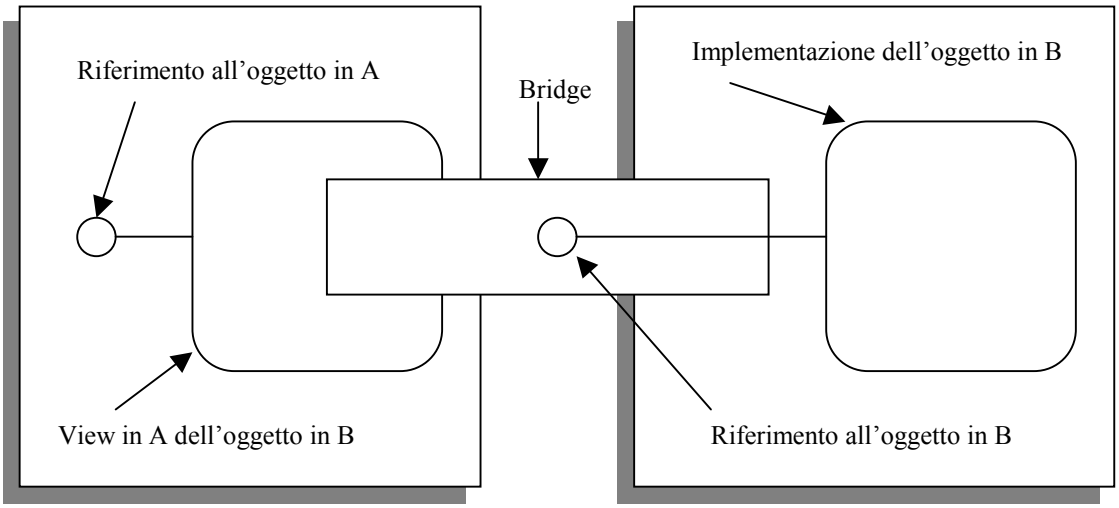


Figura 4.2 Modello di interazione B/A.

Nella parte sinistra della Figura 4.2 c'è il client del sistema A che vuole mandare una richiesta all'oggetto del sistema B, sulla destra. Ci riferiremo all'intera entità concettuale che fornisce il mapping con il nome di **Bridge** (ponte). Lo scopo è mappare e inviare trasparentemente qualsiasi richiesta dal client all'oggetto server.

Per far ciò, per prima cosa forniamo un oggetto del sistema A chiamato **View** (vista). La View è un oggetto del sistema A che ha l'identità e l'interfaccia dell'oggetto server del sistema B mappato sul sistema A, ed è descritto come una A View dell'oggetto server del sistema B.

La View espone un'interfaccia, chiamata **interfaccia View**, che è isomorfa all'interfaccia dell'oggetto del sistema B. I metodi dell'interfaccia View convertono le richieste dei client del sistema A in richieste all'interfaccia dell'oggetto del sistema B. La View è un componente del Bridge. Un Bridge può essere composto da più View.

Il Bridge mappa interfacce e identifica moduli tra sistemi differenti. Concettualmente, il Bridge mantiene un riferimento all'oggetto del sistema B (comunque questo non è fisicamente richiesto). Il Bridge deve fornire un punto di incontro tra il sistema A e il sistema B e potrebbe essere implementato usando qualsiasi meccanismo che permette

la comunicazione tra due sistemi (IPC, RPC, rete, memoria condivisa, ecc.) sufficiente per conservare la semantica rilevante dell'oggetto che viene mappato.

I client trattano la View come un oggetto reale del sistema A e formulano le loro richieste nel modo tipico del sistema A. La richiesta è tradotta nella corrispondente richiesta del sistema B e successivamente inviata all'oggetto server. L'effetto finale è che una richiesta fatta ad un interfaccia nel sistema A è trasparentemente inviata ad un oggetto del sistema B.

Questo modello di integrazione funziona in entrambe le direzioni. Per esempio, se il sistema A è COM e il sistema B è CORBA, allora la View è chiamata la View COM di un oggetto CORBA. La COM View presenta al client COM l'interfaccia dell'oggetto CORBA. Viceversa, se il sistema A è CORBA e il sistema B è COM, allora la View è chiamata la View CORBA dell'oggetto COM. La CORBA View presenta al client CORBA l'interfaccia dell'oggetto COM. Quindi le possibili interazioni sono:

- Il mapping che fornisce una COM View di un oggetto CORBA.
- Il mapping che fornisce una CORBA View di un oggetto COM.

La divisione del processo di mapping nei componenti visti in precedenza non implica nessuna strategia particolare dell'implementazione. Per esempio, una COM View e il suo riferimento incapsulato possono essere implementati in COM come un singolo componente oppure come un sistema di componenti comunicanti installati su differenti macchine.

Questa architettura è aperta ad ogni strategia di implementazione, inclusa, ma non limitata, al mapping generico o specifico di un'interfaccia. Analizziamo i due casi in dettaglio:

- Il **Mapping Generico** consiste nel mappare tutte le interfacce attraverso un meccanismo dinamico fornito a **runtime** da un singolo insieme di componenti Bridge. Questo permette l'accesso automatico alle nuove interfacce, appena esse sono registrate nel sistema server. Questo approccio semplifica l'installazione e la gestione dei cambiamenti, ma può incorrere in prestazioni scadenti tipiche del collegamento dinamico.
- Il **Mapping Specifico** di un'interfaccia consiste invece nel generare componenti Bridge separati per ogni interfaccia, oppure per un limitato insieme di interfacce in relazione tra loro, per esempio da un compilatore. Questo approccio è molto buono dal punto di vista dalle prestazioni, ma può creare problemi nell'installazione e nella gestione di nuovi componenti.

4.3 I problemi del mapping

Lo scopo della specifica CORBA che tratta dell'interazione tra vari sistemi è ottenere un semplice mapping bidirezionale (nel nostro caso CORBA/COM e COM/CORBA)

secondo lo schema visto in precedenza. Comunque, a dispetto di molte similitudini, ci sono molte differenze significative tra CORBA e COM che complicano il raggiungimento di questo scopo. Le più importanti sono le seguenti:

- Il **mapping dell'interfaccia**. Un'interfaccia CORBA deve essere mappata su due diverse forme di interfacce, OLE Automation e COM, e viceversa.
- Il **mapping della composizione di interfacce**. L'ereditarietà multipla di CORBA deve essere mappata in un misto di ereditarietà singola e aggregazione di COM. Dall'altra parte, l'aggregazione di COM deve essere mappata in un modello di ereditarietà multipla di CORBA.
- Il **mapping dell'identità**. La nozione esplicita dell'identità di un'istanza in CORBA deve essere mappata nella più implicita nozione di identità in COM.
- L'**invertibilità del mapping**. Potrebbe essere desiderabile che il mapping di un oggetto fosse reversibile, ma la specifica non garantisce l'invertibilità in tutte le situazioni.

4.4 Il mapping dell'interfaccia

Lo standard CORBA per descrivere interfacce è l'OMG IDL. Esso descrive le richieste che un oggetto supporta. Come abbiamo già visto, OLE fornisce due distinti, e a volte disgiunti, modelli di interfacce: COM e Automation. Ognuna ha le sue proprie forme di richiesta, la sua semantica e la sua sintassi.

Quindi dobbiamo considerare i problemi e i benefici di quattro forme di mapping:

- CORBA/COM
- CORBA/Automation
- COM/CORBA
- Automation/CORBA

Dobbiamo anche considerare l'impatto bidirezionale di una terza e ibrida forma di interfaccia, l'interfaccia **dual**, che supporta sia le interfacce Automation che quelle COM.

Vediamo adesso una breve descrizione e i principali problemi che si incontrano in ognuna di queste forme di mapping.

4.4.1 CORBA/COM

C'è un buon mapping da oggetti CORBA a interfacce COM; infatti:

- Le primitive OMG IDL hanno quasi tutte una corrispettiva primitiva COM.
- I tipi di dato strutturati (unioni, strutture, array, stringhe ed enumerati) sono mappati completamente.

- I riferimenti ad oggetti CORBA hanno il loro corrispettivo nei puntatori a interfacce COM.
- Le interfacce ereditate CORBA possono essere rappresentate come interfacce multiple COM.
- Gli attributi CORBA possono essere mappati come due operazioni (set e get) delle interfacce COM.

Questo mapping è forse il modo più naturale per rappresentare le interfacce di oggetti CORBA in un ambiente COM. In pratica, comunque, molti client COM (per esempio quelli che usano linguaggi di scripting) si possono solo collegare alle interfacce Automation e non a quelle più generali COM. Perciò, fornire solo il mapping CORBA/COM non sarebbe sufficiente per soddisfare tutti i client COM/OLE.

4.4.2 CORBA/Automation

C'è un'equivalenza limitata tra oggetti OLE Automation e oggetti CORBA:

- Alcune delle primitive OMG IDL si mappano direttamente sulle primitive Automation. Comunque, ci sono primitive in entrambi i sistemi (per esempio il tipo OLE **CURRENCY** e i tipi interi **unsigned** di CORBA) che devono essere mappati come casi speciali (perdendo molte volte alcuni aspetti della loro semantica iniziale).
- I tipi strutturati di OMG IDL non hanno nessun corrispettivo naturale nei costrutti di Automation. Poiché i tipi strutturati non possono essere passati come parametri nelle interfacce Automation, devono essere simulati tramite speciali interfacce (per esempio si può vedere una struttura come un'interfaccia Automation che ha solo i metodi set e get per poter accedere ai propri dati membro).
- Le **Repositories** delle interfacce CORBA possono essere mappate dinamicamente con le **Type Library** di Automation.
- I riferimenti ad oggetti CORBA si mappano con i puntatori alle interfacce di Automation.
- Non c'è un modo *pulito* di mappare l'ereditarietà multipla sulle interfacce Automation. Tutti i metodi delle interfacce che hanno un'ereditarietà multipla devono essere inseriti in una singola interfaccia Automation; comunque, questo approccio richiede un ordinamento totale dei metodi, se si vogliono supportare le interfacce duali. Un approccio alternativo è quello di mappare l'ereditarietà multipla con le interfacce multiple. Questo mapping, comunque, suppone che si debba fornire un meccanismo per poter cambiare interfaccia dato che la tradizionale **QueryInterface** non funziona con gli oggetti OLE.
- Gli attributi CORBA possono essere mappati con le proprietà (set e get) delle interfacce Automation.

Questa forma di mapping, anche se pone alcune restrizioni, è il modo più naturale per fornire un accesso dinamico agli oggetti CORBA da parte dei client OLE Automation.

4.4.3 COM/CORBA

Questo mapping è simile a quello CORBA/COM, eccetto per i punti seguenti:

- Alcune unioni, alcuni puntatori e il tipo **SAFEARRAY**, richiedono un trattamento speciale.
- Sebbene sia meno comune, gli oggetti COM possono essere creati direttamente in C e C++ (senza esporre una specifica interfaccia) implementando il **marshalling** in forma proprietaria. Se l'interfaccia può essere definita precisamente in un qualche formalismo COM (MIDL, ODL o Type Library), si deve fare questa traduzione manualmente prima di poter fare qualsiasi forma di mapping. Se non è possibile, si deve fornire anche una forma proprietaria di mapping.
- MIDL, ODL e le Type Library sono formalismi piuttosto differenti e non sono tutti supportati su certe piattaforme Windows (per esempio MIDL non è supportato sull'ormai obsoleta piattaforma Win16).

4.4.4 Automation/CORBA

Poiché le interfacce e i tipi di dato di Automation sono notevolmente vincolati, i problemi del mapping tra Automation e CORBA sono limitati. Infatti:

- Le interfacce Automation e i loro riferimenti (puntatori a **IDispatch**) si mappano direttamente con le interfacce e i riferimenti agli oggetti CORBA.
- Le sigle delle richieste Automation si mappano direttamente con le sigle delle richieste CORBA.
- Molti dei tipi di dato Automation si mappano direttamente sui tipi di dato CORBA. Alcuni tipi (come ad esempio **CURRENCY**), non hanno un corrispettivo in CORBA, ma possono essere facilmente mappati in tipi strutturati a loro isomorfi.
- Le proprietà Automation si mappano con gli attributi CORBA.

4.5 Il mapping delle interfacce composte

CORBA fornisce un modello di ereditarietà multipla per aggregare ed estendere le interfacce degli oggetti. Di conseguenza le interfacce CORBA sono essenzialmente definite staticamente sia in OMG IDL che nell'Interface Repository. L'evoluzione a **runtime** delle interfacce è possibile derivando nuove interfacce da quelle già esistenti. Qualsiasi riferimento ad un oggetto CORBA punta ad un oggetto che espone, in qualsiasi momento, un'interfaccia singola (la più derivata) nella quale sono inglobate tutte le sue interfacce di base. Il modello CORBA non supporta gli oggetti che hanno interfacce multiple e distinte.

Dall'altra parte, gli oggetti COM espongono interfacce aggregate fornendo un meccanismo uniforme per navigare tra le interfacce che un singolo oggetto supporta (il

metodo QueryInterface). Inoltre, COM suppone che l'insieme di interfacce che un oggetto supporta cambierà a runtime. Il solo modo per sapere se un oggetto supporta un'interfaccia in un particolare momento è quello di interrogare l'oggetto stesso.

Gli oggetti OLE Automation tipicamente forniscono tutte le operazioni Automation in una singola e *appiattita* interfaccia IDispatch. Un meccanismo analogo a QueryInterface è fornito anche da OLE (tramite le interfacce duali), ma siccome non tutti i client OLE supportano questo meccanismo (i linguaggi di scripting ad esempio), non c'è un modo standard per navigare tra le interfacce.
Vediamo adesso il mapping delle interfacce composte nei differenti casi possibili.

4.5.1 CORBA/COM

L'ereditarietà multipla di CORBA è mappata con qualche difficoltà nelle interfacce COM. Dall'esperienza della programmazione orientata agli oggetti si desume che ci sono due tipi di ereditarietà, l'**estensione** e l'**unione**. L'ereditarietà può essere usata per estendere un'interfaccia linearmente, creando una specializzazione o una nuova versione dell'interfaccia ereditata. Ma l'ereditarietà (in particolare l'ereditarietà multipla) è anche comunemente usata per aggiungere una nuova potenzialità che può essere ortogonale alle altre funzioni dell'oggetto (per esempio l'ereditarietà multipla di due interfacce, una delle quali permette la memorizzazione e l'altra la visualizzazione).

Idealmente, l'estensione viene mappata in modo adeguato nel modello di ereditarietà singola producendo una catena lineare di interfacce. Questo uso dell'ereditarietà CORBA per la specializzazione si mappa direttamente su COM; un'unica catena di interfacce ereditate CORBA si mappa su una singola *vtable* di interfaccia COM che include tutti gli elementi delle interfacce CORBA presenti nella catena. È comunque necessario ordinare i metodi delle interfacce ereditate per fornire un metodo deterministico per mappare le interfacce CORBA sulle vtable COM.

L'uso dell'ereditarietà per unire due o più potenzialità si mappa adeguatamente nel meccanismo dell'aggregazione di COM; ogni interfaccia che eredita da più interfacce è mappata su un'interfaccia COM singola separata che aggrega le altre funzionalità (che possono essere richieste tramite l'invocazione di QueryInterface con l'appropriato UUID.

Sfortunatamente, con l'ereditarietà multipla di CORBA non c'è un modo sintattico per determinare se una particolare ereditarietà è un'estensione o un'unione. Perciò non è possibile avere un meccanismo di mapping ideale. Il metodo che vedremo successivamente quando ci occuperemo del mapping dettagliato, cerca di giungere ad un compromesso tra l'equivalenza della semantica e la facilità della gestione. Esso soddisfa il primo requisito per l'integrazione perché descrive un mapping uniforme e

deterministico tra qualsiasi grafo di ereditarietà CORBA e un insieme composto di interfacce COM.

4.5.2 COM/CORBA

Le caratteristiche del modello di aggregazione delle interfacce COM possono essere conservate in CORBA fornendo un insieme di interfacce che possono essere usate per gestire una collezione di oggetti CORBA multipli con differenti e disgiunte interfacce come una singola unità composta. Vedremo successivamente il mapping nei particolari.

4.5.3 CORBA/Automation

OLE Automation (come si può vedere dall'interfaccia IDispatch) non dipende dall'ordine delle funzioni virtuali. L'oggetto server implementa l'interfaccia IDispatch come un mini interprete ed espone tutte le sue funzionalità tramite una singola interfaccia. Non è necessario quindi fornire un ordine per le operazioni che supporta.

Un problema di ordinamento rimane, comunque, per le interfacce duali. Infatti, queste sono interfacce COM le cui operazioni hanno le restrizioni dei tipi di dato Automation. Poiché restano, nonostante ciò, delle interfacce COM, un client può chiamare un metodo specifico tramite la vtable. In questo modo si scavalca il meccanismo di IDispatch ed è quindi necessario avere un ordine tra i metodi come visto in precedenza.

4.5.4 Automation/CORBA

Le interfacce OLE Automation sono una semplice collezione di operazioni, senza ereditarietà o aggregazione. Quindi ogni interfaccia IDispatch si mappa direttamente su un'interfaccia CORBA equivalente.

4.6 Il mapping nei dettagli

Occupiamoci adesso del mapping nei dettagli per tutti i casi presi in considerazione dei paragrafi precedenti.

4.6.1 Regole di ordinamento CORBA/MIDL

- Ogni interfaccia OMG IDL che non ha un'interfaccia madre è mappata su un'interfaccia MIDL che deriva da IUnknown.
- Ogni interfaccia OMG IDL che eredita da una singola interfaccia madre è mappata su un'interfaccia MIDL che deriva dal mapping dell'interfaccia madre.
- Ogni interfaccia OMG IDL che eredita, in modo multiplo, da più interfacce è mappata su un'interfaccia MIDL che deriva da IUnknown (l'oggetto aggregherà le altre interfacce).
- Per ogni interfaccia CORBA, il mapping delle operazioni precede quello degli attributi.
- Il mapping risultante delle operazioni di un'interfaccia è ordinato in base al nome delle operazioni. L'ordine è lessicografico.
- Il mapping risultante degli attributi di un'interfaccia è ordinato in base al nome degli attributi. L'ordine è lessicografico. Se l'attributo non è **readonly**, il metodo **_get_** precede immediatamente il metodo **_put_**.

4.6.2 Regole di ordinamento CORBA/OLE

- Ogni interfaccia OMG IDL che non ha un'interfaccia madre è mappata su un'interfaccia ODL che deriva da IDispatch.
- Ogni interfaccia OMG IDL che eredita da una singola interfaccia madre è mappata su un'interfaccia ODL che deriva dal mapping dell'interfaccia madre.
- Ogni interfaccia OMG IDL che eredita, in modo multiplo, da più interfacce è mappata su un'interfaccia ODL che deriva, usando l'ereditarietà singola, dal mapping della prima interfaccia madre. La prima interfaccia madre è definita come la prima interfaccia quando le interfacce madri sono ordinate in base al loro **repository id**. L'ordine è lessicografico.
- In un'interfaccia il mapping delle operazioni precede il mapping degli attributi.
- Le operazioni sono ordinate, nel mapping risultante, in base al loro nome. L'ordine è lessicografico.
- Gli attributi sono ordinati, nel mapping risultante, in base al loro nome. L'ordine è lessicografico. Per gli attributi che non sono readonly, il metodo **[propget]** precede immediatamente il metodo **[propput]**.
- Per le interfacce OMG IDL che ereditano, in modo multiplo, da più interfacce, le operazioni introdotte nell'interfaccia corrente sono mappate per prime e sono ordinate in base alle regole precedenti. Successivamente, si inseriscono le operazioni (sempre ordinate in base alle regole precedenti) che scaturiscono dal mapping delle operazioni delle interfacce madri (esclusa la prima che è mappata usando l'ereditarietà).

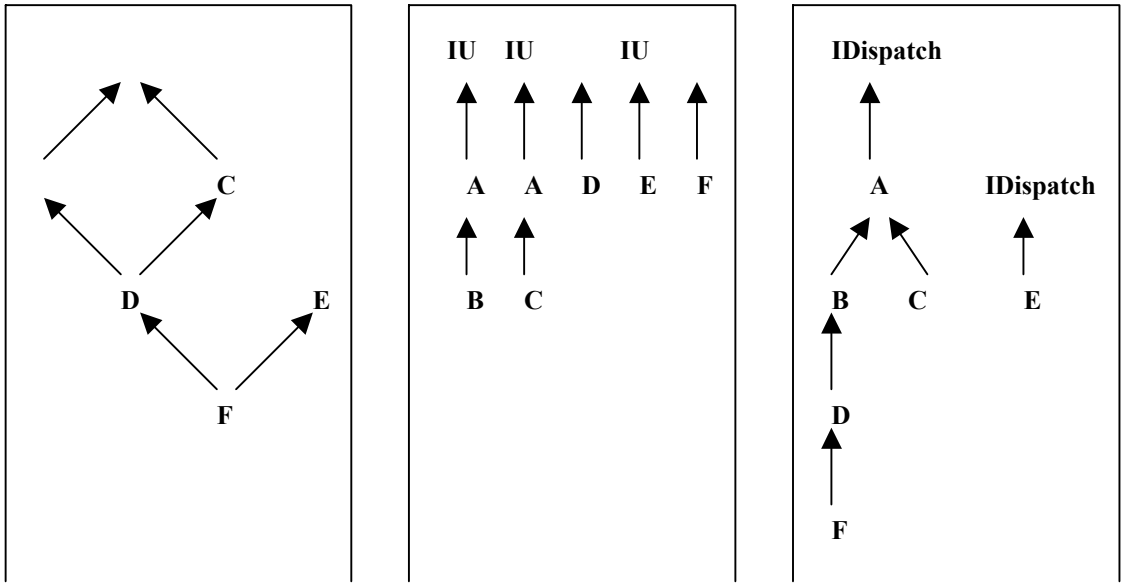
4.6.3 Esempio di mapping

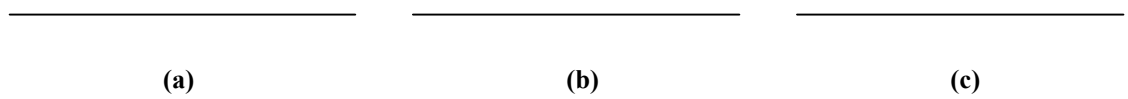
Per chiarire meglio le regole viste in precedenza vediamo un esempio di mapping sia del caso CORBA/MIDL che del caso CORBA/OLE.

Consideriamo il seguente listato OMG IDL:

```
interface A {
   void opA();
   attribute long val;
};

interface B : A {
   void opB();
};

interface C : A {
   void opC();
};

interface D : B, C {
   void opD();
};

interface E {
   void opE();
};

interface F : D, E {
   void opF();
};
```

Da questo listato si ottiene il grafo mostrato nella Figura 4.3 (a).





(a) **(b)** **(c)**

Figura 4.3 Grafo delle derivazioni CORBA (a) e delle corrispondenti MIDL (b) e OLE (c).

Seguendo le regole CORBA/MIDL otteniamo il seguente listato:

```
[object, uuid(…)]
interface IA : IUnknown {
   HRESULT opA();
   HRESULT get_val([out] long* val);
   HRESULT put_val([in] long val);
};

[object, uuid(…)]
interface IB : IA {
   HRESULT opB();
};

[object, uuid(…)]
interface IC : IA {
   HRESULT opC();
};

[object, uuid(…)]
interface ID : IUnknown {
   HRESULT opD();
};

[object, uuid(…)]
interface IE : IUnknown {
   HRESULT opE();
};

[object, uuid(…)]
interface IF : IUnknown {
   HRESULT opF();
};
```

Dall'altra parte, seguendo le regole CORBA/OLE otteniamo:

```
[oleautomation, dual, uuid(…)]
interface DA : IDispatch {
   HRESULT opA([out, optional] VARIANT* v);
   [propget] HRESULT val([out] long* val);
   [propput] HRESULT val([in] long val);
};

[oleautomation, dual, uuid(…)]
```

```
interface DB : DA {
   HRESULT opB([out, optional] VARIANT* v);
};

[oleautomation, dual, uuid(…)]
interface DC : DA {
   HRESULT opC([out, optional] VARIANT* v);
};

[oleautomation, dual, uuid(…)]
interface DD : DB {
   HRESULT opD([out, optional] VARIANT* v);
   HRESULT opC([out, optional] VARIANT* v);
};

[oleautomation, dual, uuid(…)]
interface DE : IDispatch {
   HRESULT opE([out, optional] VARIANT* v);
};

[oleautomation, dual, uuid(…)]
interface DF : DD {
   HRESULT opF([out, optional] VARIANT* v);
   HRESULT opE([out, optional] VARIANT* v);
};
```

4.7 Il mapping dell'identità dell'interfaccia

Una specifica di integrazione abbastanza generale deve permettere l'integrazione di componenti sviluppati da diversi produttori (per esempio, una COM View creata dal produttore A può invocare un server CORBA del produttore B di cui condivide la stessa interfaccia IDL). Per permettere ciò, tutte le COM View mappate da una particolare interfaccia CORBA devono condividere lo stesso identificatore di interfaccia COM. In questa sezione vedremo un mapping uniforme dal Repository ID di CORBA all'ID di interfaccia COM.

4.7.1 Dal Repository ID al COM ID

Un Repository ID di CORBA è mappato su un corrispondente ID di COM usando un derivato dell'algoritmo **MD5 Message-Digest** del **RSA Data Security**. Il Repository ID di un'interfaccia CORBA è trasformato dall'algoritmo **MD5** in un identificatore *hash* a 128 bit. Il byte meno significativo è il byte 0 e quello più significativo è il byte 8. I risultanti 128 bit sono modificati come segue.

Lo spazio **DCE** dell'UUID è correntemente diviso in quattro gruppi principali. Vediamo il significato del byte 8:

- 0xxxxxxx è lo spazio dei nomi **NCS1.4**
- 10xxxxxx è lo spazio dei nomi **DCE1.0**
- 110xxxxx è usato dalla Microsoft
- 1111xxxx è lasciato per sviluppi futuri

Per quanto riguarda lo spazio dei nomi **NCS1.4**, gli altri bit del byte 8 specificano una particolare famiglia. La famiglia 29 sarà assegnata per assicurare che gli ID generati automaticamente non interferiranno con le altre tecniche di generazione degli UUID.

I due bit più significativi del byte 9 hanno questo significato:

- 00 nessun significato
- 01 è un COM ID
- 10 è un Automation ID
- 11 è un ID di un'interfaccia dual di Automation

Questi bit non devono essere mai usati per determinare il tipo di interfaccia. Sono usati solo per evitare collisioni dei nomi quando si generano gli ID per i diversi tipi di interfacce dual, COM o Automation.

I rimanenti bit sono presi dall'algoritmo MD5 (e sono memorizzati nell'UUID con l'ordine **little endian**).

L'ID COM generato da un repository ID di CORBA sarà usato da una COM View di un'interfaccia CORBA, eccetto quando il repository ID è un DCE UUID e l'ID che sarà generato è per un'interfaccia COM (non Automation o dual). In questo caso il DCE UUID sarà usato lui stesso come ID COM invece di generarlo dal corrispondente repository ID (questo per permettere agli sviluppatori di server CORBA di implementare le interfacce COM esistenti).

Questo meccanismo non richiede nessun cambiamento all'IDL. Comunque c'è l'assunzione implicita che il repository ID deve essere unico tra gli **ORB** per interfacce differenti e identico (sempre tra gli ORB) per le stesse interfacce.

Questa assunzione è anche necessaria per l'**IIOP** (**Internet Inter ORB Protocol**) per funzionare correttamente tra i vari ORB.

4.7.2 Dal COM ID al Repository ID

Il mapping di un COM ID su un ID di interfaccia CORBA è specifico di ogni venditore. Comunque, deve essere lo stesso se il mapping CORBA di un'interfaccia COM è stato fatto con la direttiva *#pragma ID<nome_interfaccia> = "DCE:..."*.

Vediamo un esempio con la definizione MIDL che segue:

```
[uuid(f4f2f07c-3a95-11cf-affb-08000970dac7), object]
interface A : IUnknown {
...
};
```

viene mappata con questo frammento di OMG IDL:

```
interface A {
#pragma ID A = "DCE:f4f2f07c-3a95-11cf-affb-08000970dac7"
...
};
```

4.8 Identità, *binding* e ciclo di vita

Il modello di integrazione illustrato nella Figura 4.2 mappa una View di un sistema con un riferimento dell'altro sistema. Questa relazione fa nascere alcune domande:

- Come i concetti di **identità** e **ciclo di vita** di un oggetto nei due sistemi possono corrispondere e, data la loro notevole differenza, come possono essere appropriatamente mappati?
- Come è legata la View di un sistema ad un riferimento (ad un oggetto) nell'altro sistema?

Queste domande nascono dal fatto che COM e CORBA hanno delle differenti nozioni di cosa significa identità e ciclo di vita di un oggetto. L'impatto di queste differenze tra i due modelli ha delle conseguenze sulla trasparenza del mapping.

Nei paragrafi successivi approfondiremo questi temi.

4.8.1 Identità degli oggetti CORBA

CORBA definisce un oggetto come una combinazione di stato e insieme di metodi che esplicitamente incarnano un'astrazione caratterizzata dal comportamento delle relative richieste. Il riferimento ad un oggetto è definito come un nome che attendibilmente e consistentemente denota un particolare oggetto. Una descrizione pratica di un particolare oggetto in termini CORBA è un'identità che espone una coerenza dell'interfaccia, del comportamento e dello stato durante il suo ciclo di vita. Questa descrizione può cadere in fallo in alcuni casi limite, ma sembra essere abbastanza ragionevole per dare una nozione intuitiva di identità di un oggetto.

Altre proprietà importanti degli oggetti CORBA sono:

- Gli oggetti hanno delle identità nascoste che sono incapsulate nei loro riferimenti.

- Le identità degli oggetti sono uniche in un particolare dominio che è almeno il dominio di un ORB.
- I riferimenti denotano affidabilmente un particolare oggetto. Possono essere usati per identificare e localizzare un particolare oggetto allo scopo di inviargli una richiesta.
- Le identità sono immutabili e persistenti per tutta la vita dell'oggetto a cui si riferiscono.
- I riferimenti possono essere usati come destinazione delle richieste senza tenere conto dello stato e della locazione dell'oggetto che denotano; se un oggetto è passivamente memorizzato quando un client effettua una richiesta ad un suo riferimento, l'ORB è responsabile della sua localizzazione e attivazione in modo del tutto trasparente.
- Non c'è nessuna nozione di *connessione* tra riferimento ed oggetto e non c'è nessuna nozione di **reference counting**.
- I riferimenti possono essere espressi come stringhe e reinterpretati ovunque (sempre entro il dominio dell'ORB).
- Due riferimenti possono essere testati per l'equivalenza, cioè possono essere confrontati per vedere se si riferiscono alla stessa istanza di un oggetto, anche se solo il risultato TRUE può essere considerato attendibile.

4.8.2 Identità degli oggetti COM

La nozione di cosa significa essere "un particolare oggetto COM" è meno definita di quella CORBA. In pratica, questa nozione tipicamente corrisponde ad un'istanza attiva di un'implementazione, ma non un particolare stato persistente. Un'istanza COM può essere molto più precisamente definita come "l'entità la cui interfaccia (o invece, una delle sue interfacce) è restituita dall'invocazione della funzione **IClassFactory::CreateInstance**".

Per quanto riguarda le istanze di oggetti COM, si possono fare queste osservazioni:

- Le istanze COM sono inizializzate con uno stato di default vuoto (ad esempio un documento con nessun contenuto) o sono inizializzate con uno stato arbitrario. La funzione **IClassFactory::CreateInstance** non ha parametri per settare uno stato iniziale.
- I soli riferimenti possibili per le istanze COM sono i puntatori alle sue interfacce. La loro utilità per determinare l'equivalenza di oggetti è limitata allo *scope* del processo in cui si trovano. Ogni oggetto COM può stabilire la sua nozione di identità persistente, ma questa non fa parte della specifica COM.
- Non c'è nessun meccanismo inerente per determinare se due puntatori a interfaccia si riferiscono allo stesso oggetto.
- L'identità e la gestione dello stato sono generalmente indipendenti dall'identità e dal ciclo di vita delle istanze COM. Ad esempio i file che contengono lo stato di un documento sono persistenti, e sono identificati all'interno dello spazio dei nomi del File System. Una singola istanza COM di tipo documento può caricare, manipolare

e memorizzare molti documenti durante la sua vita; un singolo documento può essere caricato e usato da molti oggetti che hanno possibilmente anche un differente tipo. Ogni relazione tra uno oggetto COM e uno stato è un concetto artificiale che appartiene a quel particolare tipo di oggetto.

4.8.3 Binding e ciclo di vita

I problemi relativi all'identità discussi in precedenza fanno nascere dei problemi pratici quando si definisce il meccanismo di gestione del binding e del ciclo di vita degli oggetti. Il binding si riferisce al modo col quale un oggetto di un sistema può essere localizzato da un client dell'altro sistema e all'associazione di un'appropriata View. Il ciclo di vita, in questo contesto, si riferisce al modo in cui gli oggetti vengono creati e distrutti dai client.

4.8.3.1 Ciclo di vita

La persistenza in memoria degli oggetti COM (inclusi quelli Automation) è legata al ciclo di vita dei propri client. Questo significa, in COM, che un oggetto è distrutto quando non ci sono più client attaccati ad esso. Se rimangono dei client attivi, l'oggetto non può essere rimosso dalla memoria. Sfortunatamente, il meccanismo di **reference counting** usato da COM ha dei problemi quando è applicato ad una rete su un'area molto vasta, quando il traffico di rete è elevato e quando la rete e i **router** non sono **fault tolerant** (o non affidabili al 100%). Per esempio, se la rete che connette i client e i server non è in funzione, il server potrebbe pensare che i suoi client sono terminati e terminerebbe a sua volta (se non ci sono dei client locali). Quando la connessione sarà riattivata, anche pochi secondi dopo, i client avranno dei riferimenti ad oggetti non più validi e dovranno ripartire (oppure dovranno essere predisposti a gestire una situazione del genere tramite il meccanismo delle eccezioni). Inoltre, se ci sono dei client attivi per un oggetto ma che lo usano raramente, il server deve rimanere in memoria. In un sistema molto ampio che ha molti server, questo meccanismo di gestione della memoria non è certo la cosa migliore che si può ottenere.

Dall'altra parte, nel modello CORBA il ciclo di vita dei client è sdoppiato da quello dei server. Il modello CORBA permette ai client di mantenere dei riferimenti agli oggetti anche quando non sono più in funzione. I server possono disattivarsi e distruggersi da soli quando non ci sono più client che *correntemente* li stanno usando. Questo comportamento evita i problemi e le limitazioni introdotte dal reference counting distribuito. I client possono partire e terminare senza comportare un massiccio caricamento di dati da parte dei server. I server possono liberare la memoria (ma devono essere ricaricati successivamente) se non sono stati usati recentemente o se la connessione non è attiva. In aggiunta, poiché il ciclo di vita dei client e dei server

è sdoppiato, in CORBA non c'è bisogno del meccanismo del reference counting che è molto costoso se è applicato su una rete locale e impraticabile su una rete più vasta.

4.8.3.2 Binding tra oggetti CORBA e COM View

COM e Automation hanno un meccanismo limitato per registrare ed accedere agli oggetti attivi. Una singola istanza di una classe COM può essere registrata nel **registry**. I client COM e Automation possono ottenere un puntatore all'interfaccia IUnknown di un oggetto attivo con la funzione **GetActiveObject** (di COM) o con la funzione **GetObject** (di Automation). Il modo più naturale per i client COM e Automation per accedere ad un oggetto CORBA esistente è attraverso questo meccanismo o uno simile.

Le soluzioni per l'integrazione possono, se necessario, creare una COM View per ogni oggetto CORBA e registrarla nel registry, così la View (e così l'oggetto) può essere raggiunta attraverso GetActiveObject o GetObject.

Le risorse associate con il meccanismo del registry sono limitate; alcune soluzioni per l'integrazione non saranno in grado di mappare oggetti efficientemente attraverso il registry. La specifica CORBA propone di definire un'interfaccia, **ICORBAFactory**, che permette alle soluzioni per l'integrazione di fornire il loro proprio spazio dei nomi attraverso il quale gli oggetti CORBA sono resi accessibili ai client COM e Automation. Questo meccanismo è simile a quello nativo di OLE (GetObject). La specifica CORBA propone anche altre interfacce standard che servono per fornire un meccanismo uniforme alle varie architetture di integrazione. Ci occuperemo di queste successivamente.

4.8.3.3 Binding tra oggetti COM e CORBA View

Come descritto in precedenza le istanze di classi COM sono inerentemente transitorie. Tipicamente i client gestiscono gli oggetti COM e Automation creando delle nuove istanze (tramite le **Factory**) e successivamente associando loro uno stato (persistente). L'interfaccia **SimpleFactory** proposta dalla specifica CORBA (e che vedremo nei dettagli) è stata progettata per permettere ai client CORBA di creare (e legare) oggetti COM. Per ogni **ClassFactory** di COM c'è una corrispondente interfaccia SimpleFactory di CORBA. La specifica non propone nessun metodo per mappare le due factory, quindi ogni soluzione per l'integrazione è lasciata libera di agire come meglio crede. Inoltre, non c'è nessuna indicazione su come le SimpleFactory siano rese disponibili ai client CORBA.

4.8.3.4 COM View del ciclo di vita CORBA

L'interfaccia SimpleFactory fornisce un modo per creare oggetti senza parametri. Le SimpleFactory di CORBA possono essere inglobate nelle interfacce IClassFactory di COM e registrate nel registry. Il processo per costruire, definire e registrare le factory è specifico dell'implementazione.

Per permettere agli sviluppatori di COM e Automation di beneficiare del più versatile ciclo di vita degli oggetti CORBA, si applicano le seguenti regole alle COM e Automation View di oggetti CORBA.

- Quando una COM o Automation View di un oggetto CORBA non è più usata da un client e quando non ci sono più client che la usano, essa si distrugge; non dovrebbe però distruggere anche l'oggetto CORBA a cui si riferisce.
- I client della View devono chiamare il metodo **LifeCycleObject::remove** (se l'interfaccia è supportata) per rimuovere l'oggetto CORBA, altrimenti la rimozione dell'oggetto CORBA è specifica dell'implementazione.

Attualmente COM fornisce un meccanismo ai client per controllare la persistenza degli oggetti (equivalente alla *esternalizzazione* CORBA). Comunque, a differenza di CORBA, COM non fornisce un meccanismo generale per interagire, ad esempio con i database, che sono inerentemente persistenti (e memorizzano da soli il loro stato e non certo tramite l'interfaccia IPersistentStorage di COM). COM fornisce i **Moniker** che sono concettualmente equivalenti ai riferimenti persistenti degli oggetti CORBA.

Per permettere agli sviluppatori COM di usare tutte le caratteristiche degli oggetti CORBA, la specifica CORBA definisce un meccanismo che permette ai moniker di essere usati come riferimenti persistenti ad oggetti CORBA e definisce una nuova interfaccia**, IMonikerProvider**, che permette ai client di ottenere un puntatore all'interfaccia **IMoniker** dalle COM e Automation View. Il risultato finale è quello di avere un moniker che è in grado di memorizzare e caricare la versione stringa di un riferimento ad un oggetto CORBA.

4.8.3.5 CORBA View del ciclo di vita COM

I riferimenti iniziali agli oggetti COM e Automation possono essere ottenuti in questo modo: la **IClassFactory** di COM può essere inglobata nell'interfaccia SimpleFactory di CORBA. Questa SimpleFactory View di IClassFactory di COM può essere successivamente installata nel **naming service** di CORBA. Il meccanismo per registrare e per cercare dinamicamente le factory non è coperto dalla specifica CORBA.

Tutte le CORBA View degli oggetti COM e Automation supportano l'interfaccia **LifeCycleObject**. Per distruggere una View si deve chiamare il metodo **remove** di questa interfaccia. Una volta che una CORBA View è stata istanziata, deve rimanere attiva in memoria, tranne nel caso in cui l'oggetto COM o Automation a cui si riferisce supporta l'interfaccia IMonikerProvider. In questo caso la CORBA View può essere tranquillamente disattivata e riattivata dato che memorizza in modo permanente il moniker dell'oggetto. La specifica CORBA non richiede la disattivazione e la

riattivazione delle View, ma se viene fornita deve essere fatta tramite il meccanismo dell'interfaccia IMonikerProvider.

4.9 Interfacce standard

Al fine di avere un'architettura di integrazione uguale per tutte le implementazioni e per cercare di sfruttare in un sistema le caratteristiche fondamentali dell'altro, la specifica CORBA definisce un insieme di interfacce che tutte le soluzioni di integrazione dovrebbero usare. Vediamole in dettaglio.

4.9.1 L'interfaccia SimpleFactory

Il sistema CORBA permette agli oggetti factory di essere definiti arbitrariamente. In contrasto, la **IClassFactory** di COM è limitata ad avere solo un costruttore e il relativo metodo (**CreateInstance**) non accetta argomenti che passano dei dati al processo di creazione delle istanze. L'interfaccia **SimpleFactory** permette agli oggetti CORBA di essere creati secondo il rigido modello di COM. L'interfaccia supporta anche la CORBA View delle factory COM. Ecco la sua definizione:

```
module CosLifeCycle
{
   interface SimpleFactory
   {
      Object create_object();
   };
};
```

L'interfaccia SimpleFactory fornisce un generico costruttore per creare oggetti che non hanno uno stato iniziale. Quindi gli oggetti CORBA, che possono essere creati senza uno stato iniziale, devono fornire delle factory che implementano l'interfaccia SimpleFactory.

4.9.2 L'interfaccia IMonikerProvider

Le COM e Automation View degli oggetti CORBA dovrebbero supportare l'interfaccia **IMonikerProvider**. Vediamo la sua definizione:

```
[
   object,
   uuid(ecce76fe-39ce-11cf-8e92-08000970dac7)
```

```
]
interface IMonikerProvider : IUnknown
{
   HRESULT get_moniker([out] IMoniker ** val);
};
```

Questa permette ai client COM di salvare persistentemente i riferimenti agli oggetti per usarli successivamente senza aver bisogno di tenere in memoria la View. Il **moniker** restituito da IMonikerProvider deve supportare almeno le interfacce **IMoniker** e **IPersistentStorage**. Per permettere che i moniker di oggetti CORBA creati da una certa implementazione di uno sviluppatore possano essere ripristinati da quella di uno sviluppatore diverso, il metodo **IPersist::GetClassID** deve restituire questo CLSID:

```
   {a936c802-33fb-11cf-a9d1-00401c606e79}
```

Inoltre, i dati memorizzati dall'interfaccia IPersistentStorage devono avere questo formato: i primi quattro byte devono essere nulli (0) e devono essere seguiti dalla lunghezza in byte della versione stringa dell'**IOR** (usando un intero senza segno a quattro byte memorizzato secondo la notazione little endian), successivamente ci deve essere la versione stringa dell'IOR stesso (senza il carattere *NULL* finale).

4.9.3 L'interfaccia ICORBAFactory

Tutte le soluzioni di integrazione che forniscono COM View di oggetti CORBA devono esporre l'interfaccia **ICORBAFactory**. Questa interfaccia è stata designata per supportare un meccanismo semplice e generale per creare nuove istanze di oggetti CORBA. Segue la sua definizione:

```
[
   object,
   uuid(204f6240-3aec-11cf-bbfc-444553540000)
]
interface ICORBAFactory : IUnknown
{
   HRESULT CreateObject([in] LPTSTR factoryName,
                        [out, retval] IUnknown ** val);
   HRESULT GetObject([in] LPTSTR objectName,
                     [out, retval] IUnknown ** val);
};
```

Una classe COM che implementa l'interfaccia ICORBAFactory deve essere registrata nel registry di Windows con queste informazioni.

1. **Class ID**: `{913d82c0-3b00-11cf-bbfc-444553540000}`
2. **Class ID tag**: `IID_ICORBAFactory`

3. **Program ID**: "CORBA.Factory.COM"

Questo oggetto potrebbe essere implementato come singola istanza e le richieste successive di creazione dovrebbero restituire un puntatore ad essa.

La specifica CORBA definisce anche un'interfaccia simile, **DICORBAFactory**, che supporta la creazione di oggetti CORBA da parte dei client OLE Automation. L'interfaccia DICORBAFactory è un'interfaccia Automation duale:

```
[
   object,
   uuid(204f6241-3aec-11cf-bbfc-444553540000)
]
interface DICORBAFactory : IDispatch
{
   HRESULT CreateObject([in] BSTR factoryName,
                        [out, retval] IDispatch ** val);
   HRESULT GetObject([in] BSTR objectName,
                     [out, retval] IDispatch ** val);
};
```

Un'istanza di questa classe deve essere registrata nel registry di Windows della macchina client usando il **Program ID** "CORBAFactory".

I metodi **CreateObject** e **GetObject** sono stati designati per simulare il comportamento delle omonime funzioni Visual Basic.

Il primo metodo, CreateObject, causa le seguenti azioni:

- Viene creata una COM View. Il meccanismo per crearla non è specificato. Una possibile (e preferibile) implementazione potrebbe essere quella che la View deleghi la creazione ad una factory COM registrata.
- Viene creato un oggetto CORBA e viene legato alla View. L'argomento, **factoryName**, identifica il tipo di oggetto CORBA che deve essere creato. Poiché il metodo CreateObject non accetta nessun altro parametro, l'oggetto CORBA deve essere anche esso creato da una factory che non accetta parametri, oppure deve essere la View che fornisce i parametri per la creazione internamente.
- La View legata è restituita al chiamante.

Il parametro factoryName, come abbiamo già detto, identifica il tipo di oggetto CORBA che deve essere creato e implicitamente identifica, direttamente o indirettamente, l'interfaccia supportata dalla View. In generale, la stringa factoryName ha la forma di una sequenza di identificatori separati da un punto ("."), come ad esempio *"parsonnel.record.person"*. Lo scopo di questa forma è di fornire un meccanismo che è familiare ai programmatori COM e Automation che sono abituati ai **ProgID**. La semantica specifica della risoluzione dei nomi è specifica dell'implementazione. Vediamo alcuni esempi di implementazioni:

- La sequenza contenuta nella factoryName potrebbe essere interpretata come una chiave per un localizzatore di factory basato sul **CosNameService**. Internamente la soluzione di integrazione potrebbe mappare il factoryName nell'appropriato class ID di COM per la View, creare la View e legarla all'oggetto CORBA.
- La creazione potrebbe essere delegata direttamente ad una factory COM, interpretando il factoryName come un ProgID COM. Il ProgID potrebbe riferirsi alla factory della View e sarebbe l'implementazione di quest'ultima ad invocare l'appropriata factory CORBA per creare l'oggetto.

Il metodo GetObject ha il seguente comportamento:

- Il parametro objectName è mappato dalla soluzione di integrazione in un riferimento ad un oggetto CORBA. Il meccanismo esatto per associare i nomi ai riferimenti non è specificato. Al fine di apparire familiare agli utenti COM e Automation, questo parametro dovrà avere la forma di una sequenza di identificatori separati da un punto (“.”), nello stesso modo visto per il parametro di createObject. Un'implementazione potrebbe, per esempio, scegliere di mappare il parametro objectName con un nome di una specifica implementazione del Naming Service. Alternativamente, una soluzione di integrazione potrebbe scegliere di mettere le View legate agli oggetti CORBA nel registro degli oggetti attivi e semplicemente delegare le chiamata di GetObject al registro.
- Il riferimento all'oggetto è legato ad un'appropriata COM o Automation View e restituito al chiamante.

Un altro tipo di nome che è specifico di CORBA è un nome singolo preceduto da un punto, come ad esempio *“.NameService”*. Quando il nome ha questa forma, la soluzione di integrazione deve interpretare questo identificatore (senza il punto iniziale) come un nome nell'interfaccia di inizializzazione dell'ORB. Più precisamente, il nome deve essere usato come un parametro nell'invocazione del metodo **CORBA::ORB::ResolveInitialReferences** sullo pseudo oggetto associato all'ICORBAFactory. Il riferimento all'oggetto risultante è legato ad un'appropriata COM o Automation View, che è restituita al chiamante.

4.9.4 L'interfaccia IForeignObject

Quando vengono restituiti dei riferimenti ad oggetti, soprattutto fra due differenti sistemi collegati con il meccanismo del Bridge (visto in precedenza) e questi vengono mappati con delle View (è il nostro caso), c'è il rischio di creare una catena arbitrariamente lunga di View. Per evitare questo, le View di almeno un sistema devono essere in grado di esporre il riferimento *indiretto* che gestiscono. Con questo metodo una View può capire se il riferimento che ha ottenuto è un riferimento *puro* ad un oggetto o se è a sua volta un riferimento ad un'altra View. In questo caso *bypassando* il riferimento indiretto si evita il problema delle catene.

L'interfaccia **IForeignObject** è stata creata per questo scopo. Ecco la sua definizione:

```
typedef struct {
   unsigned long cbMaxSize;
   unsigned long cbLengthUsed;
   [size_is(cbMaxSize), length_is(cbLengthUsed), unique]
      long *pValue;
} objSystemIDs;

[
   object,
   uuid(204f6242-3aec-11cf-bbfc-444553540000)
]
interface IForeignObject : IUnknown
{
   HRESULT GetForeignReference([in] objSystemIDs systemIDs,
                               [out] long *systemID,
                               [out] LPTSTR *objRef);
   HRESULT GetRepositoryId([out] RepositoryId *repositoryId);
};
```

Il primo parametro, **systemIDs**, è un vettore di valori *long* che specificano il sistema. Questi valori devono essere positivi, unici e conosciuti pubblicamente. L'OMG gestirà il rilascio di questi identificatori (richiesti dai vari produttori di ORB) per garantirne l'unicità. Il valore per il sistema CORBA è il valore *long* 1. Il vettore di systemIDs contiene la lista degli identificatori dei sistemi per i quali il chiamante è interessato ad ottenere un riferimento. L'ordine degli ID nella lista indica il grado di preferenza del chiamante. Se la View può produrre un riferimento per almeno uno dei sistemi specificati, lo indicherà tramite il secondo parametro, **systemID**. Il parametro **objRef** conterrà il riferimento all'oggetto convertito in formato stringa. Ogni sistema è responsabile di fornire un meccanismo per convertire i riferimenti in stringhe e viceversa. Per il sistema CORBA esiste il metodo **CORBA::ORB::object_to_string** per questo scopo.

4.9.5 L'interfaccia ICORBAObject

L'interfaccia **ICORBAObject** è un'interfaccia COM che è esposta dalle COM View per permettere ai client di avere accesso alle operazioni sui riferimenti agli oggetti CORBA, definite nella pseudo interfaccia **CORBA::Object**. Si può ottenere l'interfaccia ICORBAObject con il meccanismo tradizionale di QueryInterface. Vediamo la sua definizione:

```
[
   object,
   uuid(204f6243-3aec-11cf-bbfc-444553540000)
]
interface ICORBAObject : IUnknown
{
```

```
   HRESULT GetInterface([out] IUnknown ** val);
   HRESULT GetImplementation([out] IUnknown ** val);
   HRESULT IsA([in] LPTSTR repositoryID,
               [out] boolean * val);
   HRESULT IsNil([out] boolean * val);
   HRESULT IsEquivalent([in] IUnknown * obj,
                        [out] boolean * val);
   HRESULT NonExistent([out] boolean * val);
   HRESULT Hash([out] long * val);
};
```

I controlli Automation accedono alle operazioni sui riferimenti agli oggetti CORBA attraverso il metodo **GetCORBAObject** dell'interfaccia duale **DIORBObject** (che vedremo successivamente). La corrispondente interfaccia per Automation è:

```
[
   object,
   dual,
   uuid(204f6244-3aec-11cf-bbfc-444553540000)
]
interface DICORBAObject : IDispatch
{
   HRESULT GetInterface([out, retval] IDispatch ** val);
   HRESULT GetImplementation([out, retval] IDispatch ** val);
   HRESULT IsA([in] BSTR repositoryID,
               [out, retval] boolean * val);
   HRESULT IsNil([out, retval] boolean * val);
   HRESULT IsEquivalent([in] IDispatch * obj,
                        [out, retval] boolean * val);
   HRESULT NonExistent([out, retval] boolean * val);
   HRESULT Hash([out, retval] long * val);
};
```

4.9.6 L'interfaccia IORBObject

L'interfaccia **IORBObject** fornisce ai client COM e Automation l'accesso agli pseudo oggetti ORB. Questa è la sua definizione:

```
typedef struct {
   unsigned long cbMaxSize;
   unsigned long cbLengthUsed;
   [size_is(cbMaxSize), length_is(cbLengthUsed), unique]
      LPTSTR *pValue;
} CORBA_ORBObjectIdList;

[
   object,
   uuid(204f6245-3aec-11cf-bbfc-444553540000)
```

```
]
interface IORBObject : IUnknown
{
   HRESULT ObjectToString([in] IUnknown * obj,
                          [out] LPTSTR * val);
   HRESULT StringToObject([in] LPTSTR ref,
                          [out] IUnknown * val);
   HRESULT GetInitialReferences(
      [out] CORBA_ORBObjectIdList * val);
   HRESULT ResolveInitialReference([in] LPTSTR name,
                                   [out] IUnknown ** val);
};
```

Un riferimento a questa interfaccia si puo' ottenere con la chiamata:

```
ICORBAFactory::GetObject("CORBA.ORB.2")
```

I metodi di IORBObject sono implementati semplicemente delegando la chiamata alle omonime funzioni dell'oggetto CORBA.

Vediamo adesso la corrispondente interfaccia duale Automation:

```
[
   object,
   dual,
   uuid(204f6246-3aec-11cf-bbfc-444553540000)
]
interface DIORBObject : IDispatch
{
   HRESULT ObjectToString([in] IDispatch * obj,
                          [out, retval] BSTR * val);
   HRESULT StringToObject([in] BSTR ref,
                          [out, retval] IDispatch * val);
   HRESULT GetInitialReferences(
      [out, retval] SAFEARRAY(IDispatch *) * val);
   HRESULT ResolveInitialReference([in] BSTR name,
      [out, retval] IDispatch ** val);
   HRESULT GetCORBAObject([in] IDispatch * obj,
      [out, retval] DICORBAObject * val);
};
```

Un riferimento a questa interfaccia si può ottenere con la chiamata:

```
DICORBAFactory::GetObject("CORBA.ORB.2")
```

Questa interfaccia è molto simile a IORBObject, eccetto per un metodo in più che è **GetCORBAObject**, il quale restituisce il puntatore IDispatch all'interfaccia DICORBAObject associata al parametro **obj**. Questo metodo è fornito principalmente per premettere ai client Automation, che non possono invocare QueryInterface, di ottenere l'interfaccia DICORBAObject.

4.10 Convenzioni sui nomi

Il **tag** di default per l'ID dell'interfaccia della COM View deve essere:

```
IID_I<nome modulo>_<nome interfaccia>
```

Per esempio, se il nome del modulo è *"MyModule"* e il nome dell'interfaccia è *"MyInterface"*, allora il tag di default per l'ID dovrà essere:

```
IID_IMyModule_MyInterface
```

Se il modulo che contiene l'interfaccia è esso stesso contenuto in altri moduli, il tag di default deve essere:

```
IID_I<nome modulo>_..._<nome modulo>_<nome interfaccia>
```

dove il nome del modulo più esterno appare a sinistra di quelli che contiene. Estendendo l'esempio precedente, se il modulo *"MyModule"* si trova all'interno del modulo *"OuterModule"*, allora il tag sarà:

```
IID_IOuterModule_MyModule_MyInterface
```

Non è richiesto nessun tag standard per gli ID delle interfacce Automation e duali perché i programmi client, scritti in ambienti tipo Visual Basic, non usano esplicitamente gli UUID.

Le altre convenzioni della specifica CORBA indicano che i nomi delle interfacce COM delle View devono essere preceduti da una *"I"* maiuscola, quelle Automation da una *"D"* e quelle delle interfacce duali dalle lettere *"DI"*. Quindi l'esempio precedente viene trasformato in:

```
IOuterModule_MyModule_MyInterface

DOuterModule_MyModule_MyInterface

DIOuterModule_MyModule_MyInterface
```

rispettivamente per i nomi delle View delle interfacce COM, Automation e duali.

Se viene registrata una classe COM separata per ogni interfaccia della View, il **Program ID** di default per questa classe dovrà essere:

```
<nome modulo>"."..."."<nome modulo>"."<nome interfaccia>
```

Per l'esempio precedente il Program ID sarà:

```
"OuterModule.MyModule.MyInterface"
```

Allo stesso modo il tag per il CLSID dovrà avere questa forma:

```
CLSID_<nome modulo>_..._<nome modulo>_<nome interfaccia>
```

Il corrispondente CLSID dell'esempio sarà:

```
CLSID_OuterModule_MyModule_MyInterface
```

Capitolo 5

Mapping tra COM e CORBA

Questo capitolo descrive il mapping dei tipi di dato e delle interfacce da CORBA a COM secondo la specifica dell'**OMG Group**. Il mapping sarà trattato sia per quanto riguarda il MIDL che per l'ODL, date le differenze tra i due sistemi e gli ambienti di sviluppo che li usano.

Questa specifica è stata implementata in un **tool**, sviluppato dall'autore di questo documento, che è stato incluso nella distribuzione **Mico** di CORBA. Per eventuali riferimenti consultare il sito *http://www.mico.org*. Nel prossimo capitolo vedremo un esempio sul suo uso.

5.1 Il mapping dei tipi di dato

Il modello dei tipi di dato a cui fa riferimento il mapping per la piattaforma COM è quello del **MIDL**. Le interfacce COM che usano un marshalling proprietario devono essere trattate in modo diverso e comunque esulano da questa specifica. Il modello dei tipi di dato a cui fa riferimento il mapping per la piattaforma OLE Automation è quello dell'**ODL**. È da notare che anche se il MIDL e l'ODL sono presi come riferimento nella specifica CORBA, non è un requisito essenziale usarli in una soluzione d'integrazione.

In molti casi c'è una corrispondenza uno a uno tra i tipi di dato COM e CORBA. Comunque, in quei casi in cui non c'è una corrispondenza esatta, potrebbe essere necessaria una conversione a tempo di esecuzione.

5.1.1 Il mapping dei tipi di dato di base

I tipi di dato di base disponibili nell'OMG IDL si mappano sui corrispondenti tipi disponibili nel Microsoft IDL come mostrato dalla Figura 5.1.

OMG			
short	short	short	Intero con segno nell'intervallo $-2^{15}…2^{15}-1$
long	long	long	Intero con segno nell'intervallo $-2^{31}…2^{31}-1$
unsigned short	unsigned short	unsigned short	Intero senza segno nell'intervallo $0…2^{16}-1$
unsigned long	unsigned long	unsigned long	Intero senza segno nell'intervallo $0…2^{32}-1$
float	float	float	Numero IEEE a precisione singola
double	double	double	Numero IEEE a precisione doppia
char	char	char	Carattere ISO Latin-1 a 8 bit
boolean	boolean	boolean	Quantità a 8 bit limitata a 1 e 0
octet	byte	unsigned char	Quantità a 8 bit che garantisce nessuna conversione durante il trasferimento tra i due sistemi.

Figura 5.1 Corrispondenza tra i tipi di dati di base.

5.1.2 Il mapping delle costanti

Il mapping della parola chiave *const* dall'OMG IDL al MIDL è quasi identico. Le seguenti definizioni di costanti in OMG IDL:

```
// OMG IDL
   const short S = ...;
   const long L = ...;
   const unsigned short US = ...;
   const unsigned long UL = ...;
   const char C = ...;
   const boolean B = ...;
```

si mappano con le seguenti (identiche) definizioni di costanti in MIDL e ODL:

```
// MIDL e ODL
```

```
const short S = ...;
const long L = ...;
const unsigned short US = ...;
const unsigned long UL = ...;
const char C = ...;
const boolean B = ...;
```

L'unica differenza sta nel fatto che l'OMG IDL permette la definizione di costanti di tipo *float* e *double*, mentre COM non lo consente. Per questo motivo, un tool che genera codice MIDL o ODL da codice OMG IDL, deve restituire un errore quando incontra definizioni di costanti di questo tipo.

5.1.3 Il mapping degli enumerati

CORBA ha degli enumerati che non sono esplicitamente etichettati con dei valori, come invece succede con il MIDL e l'ODL. Questa differenza implica che qualsiasi linguaggio, su cui viene mappato l'IDL (come ad esempio il C++ o Java) e che permette il confronto di enumerati o la definizione di funzioni come il successore o il predecessore degli enumerati, deve conservare l'ordine della definizione OMG IDL.

La parola chiave del MIDL e dell'ODL *v1_enum* è richiesta per trasmettere gli enumerati come valori a 32 bit. La Microsoft stessa raccomanda di usarla sulle piattaforme a 32 bit per incrementare l'efficienza del *marshalling* e *demarshalling* dei dati, quando gli enumerati sono incapsulati in strutture o unioni.

Vediamo un esempio di mapping:

```
// OMG IDL
   enum A_or_B_or_C {A, B, C};
```

viene trasformato in:

```
// MIDL e ODL
   typedef [v1_enum] enum tagA_or_B_or_C {A = 0, B, C}
      A_or_B_or_C;
```

Si possono usare al massimo 2^{32} identificatori negli enumerati di CORBA, mentre in COM solo 2^{16}, quindi è possibile che quelli eccedenti vengano troncati (ma è solo un caso teorico e non pratico dato che difficilmente si usano enumerati con più di 65536 identificatori).

5.1.4 Il mapping del tipo stringa

Attualmente CORBA definisce i tipi *string* e *wstring* per rappresentare le stringhe rispettivamente a 8 e 16 bit (quest'ultimo per il formato **Unicode**). Le stringhe hanno il carattere *NULL* finale.

Il MIDL e l'ODL definiscono differenti tipi di dato che sono usati per rappresentare le stringhe, sia a 8 che a 16 bit. La Figura 5.2 mostra come mappare i tipi stringa dall'OMG IDL nei corrispondenti tipi del MIDL e dell'ODL.

			Descrizione
string	LPSTR	LPSTR	Stringhe a 8 bit con il carattere NULL finale
	char*	char*	Stringhe a 8 bit con il carattere NULL finale
wstring	LPTSTR	LPTSTR	Stringhe a 16 bit in standard Unicode con il carattere NULL finale
	wchar_t*	wchar_t*	Stringhe a 16 bit in standard Unicode con il carattere
		BSTR	Stringhe a 16 bit in standard Unicode che possono avere più caratteri NULL all'interno e hanno un contatore che indica la loro lunghezza

Figura 5.2 Corrispondenza tra i tipi di dato stringa.

Se viene passata una stringa di tipo *BSTR* che contiene più di un carattere *NULL* all'interno, ad un server CORBA, il client OLE Automation riceverà l'errore *E_DATA_CONVERSION*.

L'OMG IDL supporta due tipi differenti di stringhe: quelle limitate e quelle illimitate. Quelle limitate sono definite come delle stringhe che hanno una lunghezza massima specificata; quelle illimitate invece non hanno questo limite. Vediamo come vengono tradotte.

5.1.4.1 Il mapping delle stringhe illimitate

La definizione delle stringhe illimitate rispettivamente a 8 e 16 bit in OMG IDL è la seguente:

```
// OMG IDL
typedef string UNBOUNDED_STRING;

typedef wstring UNBOUNDED_WSTRING;
```

Vengono mappate con questa sintassi sia in MIDL che in ODL (è da notare il modificatore *unique* che indica un puntatore unico ad un carattere):

```
// MIDL e ODL
typedef [string, unique] char * UNBOUNDED_STRING;

typedef [string, unique] wchar_t* UNBOUNDED_WSTRING;
```

In altre parole definiamo un puntatore ad un vettore di caratteri che hanno il carattere *NULL* finale (è indicato dal modificatore *string*), che può essere esteso ed i cui elementi possono variare a tempo di esecuzione.

5.1.4.2 Il mapping delle stringhe limitate

Le stringhe limitate hanno un mapping leggermente differente. Vediamo anche qui un esempio sia per le stringhe a 8 bit che per quelle a 16:

```
// OMG IDL
const long N = ...;
typedef string<N> BOUNDED_STRING;

const long M = ...;
typedef wstring<M> BOUNDED_WSTRING;
```

si mappano in queste corrispondenti definizioni sia per il MIDL che per l'ODL:

```
// MIDL e ODL
const long N = ...;
typedef [string, unique] char (* BOUNDED_STRING) [N];

const long M = ...;
typedef [string, unique] wchar_t (* BOUNDED_WSTRING) [M];
```

Anche qui definiamo un puntatore ad un vettore di caratteri che hanno il carattere *NULL* finale ed i cui elementi possono cambiare a tempo di esecuzione, ma qui conosciamo la sua lunghezza a tempo di compilazione (e non può variare).

5.1.5 Il mapping delle strutture

L'OMG IDL usa la parola chiave *struct* per definire record di tipi, i quali non sono altro che coppie ordinate di valori <tipo, nome>. Una struttura definita in OMG IDL si mappa bidirezionalmente con una struttura definita in MIDL e ODL. Ogni membro della struttura si mappa in base alle regole viste e che vedremo.

Una struttura in OMG IDL i cui membri sono i tipi T0, T1, T2, ecc.

```
// OMG IDL
typedef ... T0;
typedef ... T1;
typedef ... T2;
...
typedef ... TN;
struct STRUCTURE
{
   T0 m0;
   T1 m1;
   T2 m2;
   ...
   Tn mN;
};
```

ha un'equivalente definizione di struttura in MIDL e ODL che è la seguente:

```
// MIDL e ODL
typedef ... U0;
typedef ... U1;
typedef ... U2;
...
typedef ... UN;
typedef struct
{
   U0 m0;
   U1 m1;
   U2 m2;
   ...
   UN mN;
} STRUCTURE;
```

Le strutture ricorsive si mappano nella stessa maniera. Per esempio:

```
// OMG IDL
struct A
{
   sequence<A> v1;
};
```

si mappa con:

```
// MIDL
typedef struct _A
{
   struct
   {
      unsigned long cbMaxSize;
      unsigned long cbLengthUsed;
      [size_is(cbMaxSize), length_is(cbLengthUsed), unique]
         struct A * pValue;
   } v1;
} A;
```

5.1.6 Il mapping delle unioni

L'OMG IDL stabilisce che le unioni devono essere discriminate. Il discriminante deve essere esso stesso contenuto all'interno dell'unione e deve essere un'espressione costante. Può essere una costante di tipo *long*, *short*, *unsigned long*, *unsigned short*, *char*, *wchar*, *boolean* ed *enum*. Ci può essere al massimo un caso di default che può apparire nella definizione e le etichette devono essere dello stesso tipo (o trasformate con un *casting* automatico al tipo) del discriminante.

La seguente definizione di unione discriminata in OMG IDL:

```
// OMG IDL
enum UNION_DISCRIMINATOR
{
   dChar,
   dShort,
   dLong,
   dFloat;
   dDouble
};

union DISCR_UNION
   switch(UNION_DISCRIMINATOR)
{
   case dChar: char c;
   case dShort: short s;
   case dLong: long l;
   case dFloat: float f;
   case dDouble: double d;
   default: octet v[8];
};
```

è mappata con quest'unione MIDL:

```
// MIDL
typedef enum
{
   dChar,
   dShort,
   dLong,
   dFloat,
   dDouble
} UNION_DISCRIMINATOR;

typedef union
   switch(UNION_DISCRIMINATOR DCE_d)
{
   case dChar: char c;
   case dShort: short s;
   case dLong: long l;
   case dFloat: float f;
   case dDouble: double d;
   default: byte v[8];
} DISCR_UNION;
```

Praticamente l'unica differenza sta nel fatto che nel caso del MIDL si deve mettere anche un identificatore per il discriminante (nel nostro caso è *DCE_d*).

5.1.7 Il mapping delle sequenze

L'OMG IDL definisce la parola chiave *sequence* come un vettore che ha due caratteristiche: una dimensione massima opzionale che è fissata a tempo di compilazione e una lunghezza che è determinata a tempo di compilazione. Come per le stringhe, l'OMG IDL permette di definire due tipi di sequenze: limitate e illimitate. Una sequenza è limitata se una dimensione massima è specificata, altrimenti è considerata illimitata.

5.1.7.1 Il mapping delle sequenze illimitate

Il seguente frammento di OMG IDL che definisce una sequenza illimitata del tipo T:

```
// OMG IDL
typedef ... T;
typedef sequence<T> UNBOUNDED_SEQUENCE;
```

si mappa con questa sintassi sia per il MIDL che per l'ODL:

```
// MIDL e ODL
```

```
typedef ... U;
typedef struct
{
   unsigned long cbMaxSize;
   unsigned long cbLengthUsed;
   [size_is(cbMaxSize), length_is(cbLengthUsed), unique]
      U * pValue;
} UNBOUNDED_SEQUENCE;
```

In altre parole, la codifica di una sequenza illimitata OMG IDL corrisponde ad una struttura MIDL o ODL che contiene un puntatore ad un array di tipo U, dove U è il mapping del tipo T. La struttura è necessaria per dare uno *scope* alla definizione degli attributi della sequenza.

5.1.7.2 Il mapping delle sequenze limitate

La seguente sintassi OMG IDL che definisce una sequenza limitata del tipo T che può al massimo raggiungere la dimensione N:

```
// OMG IDL
const long N = ...;
typedef ... T;
typedef sequence<T, N> BOUNDED_SEQUENCE;
```

si mappa in questo modo in MIDL o ODL:

```
// MIDL e ODL
const long N = ...;
typedef ... U;
typedef struct
{
   unsigned long cbMaxSize;
   unsigned long cbLengthUsed;
   [size_is(cbMaxSize), length_is(cbLengthUsed]
      U Value[N];
} BOUNDED_SEQUENCE;
```

5.1.8 Il mapping degli array

Gli array dell'OMG IDL hanno una lunghezza fissa e possono essere multidimensionali. La stessa cosa si può dire per gli array del MIDL o dell'ODL. Per questo motivo il mapping degli array è esatto e bidirezionale. Il tipo degli elementi dell'array si mappa con le regole viste in precedenza.

Il mapping di un array di un certo tipo T in OMG IDL corrisponde ad un array del tipo U in MIDL o ODL, dove U è il mapping del tipo T. Vediamolo con un semplice esempio:

```
// OMG IDL
const long N = ...;
typedef ... T;
typedef T MY_ARRAY[N];
```

si mappa con:

```
// MIDL o ODL
const long N = ...;
typedef ... U;
typedef U MY_ARRAY[N];
```

Il valore N può essere di qualsiasi tipo intero ed è definito come costante perché la dimensione dell'array deve essere specificata a tempo di compilazione. Il mapping degli array multidimensionali segue le stesse regole.

Bisogna notare inoltre che il tipo T e il tipo U non sempre hanno lo stesso nome; è il caso del tipo *octet* dell'OMG IDL che si mappa col tipo *byte* del MIDL.

5.1.9 Il mapping del tipo *any*

Il tipo CORBA *any* permette di specificare dei valori che possono essere di qualsiasi tipo. Non c'è nessun modo diretto e semplice per mappare questo tipo in COM. Una soluzione possibile (proposta dalla specifica CORBA) è quella di usare al suo posto questa definizione di interfaccia:

```
// MIDL
typedef [v1_enum] enum CORBAAnyDataTagEnum
{
   anySimpleValTag,
   anyAnyValTag,
   anySeqValTag,
   anyStructValTag,
   anyUnionValTag,
   anyObjectValTag
} CORBAAnyDataTag;

typedef union CORBAAnyDataUnion
   switch(CORBAAnyDataTag whichOne)
{
   case anyAnyValTag: ICORBA_Any * anyVal;
   case anySeqValTag:
```

```
   case anyStructValTag:
      struct
      {
         [string, unique] char * repositoryId;
         unsigned long cbMaxSize;
         unsigned long cbLengthUsed;
         [size_is(cbMaxSize), length_is(cbLengthUsed),
            unique] union CORBAAnyDataUnion * pVal;
      } multiVal;
   case anyUnionValTag:
      struct
      {
         [string, unique] char * repositoryId;
         long disc;
         union CORBAAnyDataUnion * value;
      } unionVal;
   case anyObjectValTag:
      struct
      {
         [string, unique] char * repositoryId;
         VARIANT val;
      } objectVal;
   case anySimpleValTag: // Tutti gli altri tipi
      VARIANT simpleVal;
} CORBAAnyData;

[
   object,
   uuid(74105f50-3c68-11cf-9588-aa0004004a09)
]
interface ICORBA_Any : IUnknown
{
   HRESULT _get_value([out] VARIANT * val);
   HRESULT _put_value([in] VARIANT val);
   HRESULT _get_CORBAAnyData([out] CORBAAnyData * val);
   HRESULT _put_CORBAAnyData([in] CORBAAnyData val);
   HRESULT _get_typeCode([out] ICORBA_TypeCode ** tc);
};
```

In altre parole, i tipi di dato che possono essere contenuti nel tipo *VARIANT* sono troppo restrittivi per rappresentare quelli che possono essere inclusi nel tipo *any*, come per esempio le strutture e le unioni. Quindi, nei casi in cui si può fare il mapping con il tipo VARIANT, lo si userà, negli altri casi si useranno delle strutture dati create appositamente (come abbiamo fatto nel codice).

Questo approccio però crea molti problemi a livello di implementazione. Infatti, si conserva solo la funzionalità e non la sintassi. Vediamo un esempio di uso del tipo *any* supponendo che il linguaggio **target** sia il C++:

```
// lato CORBA
```

```
any a;
long l = 10;
a << l;  // inserzione
long l1;
a >> l1; // estrazione
```

Come abbiamo detto in precedenza, il mapping non è diretto e semplice; così dal lato COM si ottiene una cosa del genere:

```
// lato COM
long l = 10;

VARIANT v;
::VariantInit(&v);
v.vt = VT_I4;  // specifica il tipo LONG
v.lVal = l;

ICORBA_Any pIAny = ...;
pIAny->_put_value(v); // inserzione
long l1;
VARIANT v1;
pIAny->_get_value(&v1); // estrazione
l1 = v1.lVal;
```

Come si può vedere, oltre ad ottenere qualcosa che sintatticamente è completamente differente, si perde un requisito fondamentale del mapping: la trasparenza. Il problema si può parzialmente eliminare creando delle classi *wrapper* che nascondono i dettagli ai programmatori.

5.2 Il mapping delle interfacce

Nei paragrafi successivi vedremo come mappare le interfacce e i componenti che ne fanno parte. Non c'è una corrispondenza uno a uno tra le interfacce dei due sistemi e quindi a volte si perdono delle funzionalità. Il mantenimento delle caratteristiche principali è comunque garantito.

5.2.1 Il mapping degli identificatori delle interfacce

Gli identificatori delle interfacce sono usati sia in CORBA che in COM per identificare unicamente le interfacce. Sono usati nel codice dei client per individuare le interfacce degli oggetti e per inviare o ricevere informazioni ad esse.

CORBA identifica le interfacce usando il **RepositoryID**, che è un identificatore unico. COM usa invece i **GUID**, una struttura simile al **DCE UUID**. Inoltre COM identifica

le interfacce anche con dei nomi, ma sono usati solo per convenienza e non hanno il requisito dell'unicità.

Il mapping del RepositoryId di CORBA è bidirezionale sui GUID di COM. L'algoritmo usato è già stato descritto nel Capitolo 4.

5.2.2 Il mapping delle operazioni

Le operazioni sono definite all'interno delle interfacce. Sono determinate in base al loro nome, ai parametri (se ci sono) e al valore di ritorno. Opzionalmente, l'OMG IDL (e non COM) permette di specificare anche le eccezioni che possono essere sollevate ed il contesto che può essere passato ad un oggetto come argomento implicito.

Gli attributi direzionali dei parametri dell'OMG IDL, *in*, *out* e *inout*, si mappano bidirezionalmente con quelli corrispondenti del MIDL e dell'ODL che sono *[in]*, *[out]* e *[in, out]*. Gli attributi *in* e *inout* sono usati per i parametri in entrata, mentre *out* e *inout* sono usati per i parametri in uscita. Vale la stessa cosa per i corrispondenti attributi COM. Per quanto riguarda i valori di ritorno, in OMG IDL possono essere di qualsiasi tipo definibile o *void* (nessuno), mentre in COM devono essere solo degli *HRESULT* (i valori di ritorno veri e propri vengono simulati con i parametri in uscita).

Vediamo adesso un esempio di mapping di un'interfaccia OMG IDL che contiene due operazioni nella corrispondente versione in MIDL e, successivamente, in ODL (dato che in questo caso non coincidono). L'operazione *Bank::Transfer* è un esempio di operazione che non ha nessun valore di ritorno. Invece, l'operazione *Bank::OpenAccount* restituisce un oggetto di tipo *Account* come risultato dell'operazione:

```
// OMG IDL
...
interface Account
{
   ...
};
interface AccountTypes
{
   ...
};

#pragma ID::BANK::Bank "IDL::BANK/Bank:1.2"
interface Bank
{
void Transfer(in Account Account1,
                in Account Account2,
                in float Amount);
Account OpenAccount(in float StartingBalance,
```

```
                        in AccountTypes AccountType);
};
```

Il corrispondente mapping delle operazioni in MIDL è rappresentato da questo frammento di codice:

```
// MIDL
[
   object,
   uuid(...),
   pointer_default(unique)
]
interface IAccount : IUnknown
{
   ...
};

[
   object,
   uuid(...),
   pointer_default(unique)
]
interface IAccountTypes : IUnknown
{
   ...
};

[
   object,
   uuid(682d22fb-78ac-0000-0c03-4d0000000000),
   pointer_default(unique)
]
interface IBank : IUnknown
{
HRESULT OpenAccount([in] float StartingBalance,
                        [in] AccountTypes AccountType,
                        [out] IAccount ** ppINewAccount);
HRESULT Transfer([in] IAccount * Account1,
                    [in] IAccount * Account2,
                    [in] float Amount);
};
```

Invece la corrispondente versione ODL è questa:

```
// ODL
[
   uuid(...)
]
interface IAccount : IUnknown
{
   ...
```

```
};

[
   uuid(...)
]
interface IAccountTypes : IUnknown
{
   ...
};

[
   uuid(682d22fb-78ac-0000-0c03-4d0000000000)
]
interface IBank : IUnknown
{
   HRESULT OpenAccount([in] float StartingBalance,
                [in] AccountTypes AccountType,
                [out, retval] IAccount ** ppINewAccount);
   HRESULT Transfer([in] IAccount * Account1,
                    [in] IAccount * Account2,
                    [in] float Amount);
};
```

L'ordine e il nome dei parametri in MIDL e ODL è identico a quello specificato nella definizione delle operazioni in OMG IDL. Nel mapping le operazioni sono ordinate lessicograficamente in base al loro nome.
È importante notare che non c'è una corrispondenza tra i parametri ed il valore di ritorno nei due sistemi. In particolare, il valore di ritorno di un'operazione definita in OMG IDL deve essere mappato come un parametro in uscita che appare alla fine nella lista dei parametri. Questo formato permette alle operazioni di avere un aspetto più naturale per i programmatori COM. Quando il valore di ritorno è trasformato in un parametro in uscita, l'operazione restituisce un *HRESULT* (come del resto tutte le operazioni COM). Senza questo meccanismo non ci sarebbe nessun modo per COM per restituire delle eccezioni ai client. Il valore dell'HRESULT è determinato in base al mapping del valore delle eccezioni CORBA (se si sono verificate).

L'*UUID* usato per l'interfaccia dal lato COM è quello che viene generato a partire dal RepositoryId dall'algoritmo MD5 descritto nel Capitolo 4.

La corrispondente interfaccia MIDL è inoltre marcata con l'attributo *unique* dei puntatori. Questo significa (se non diversamente specificato) che non ci possono essere degli *alias* per questa interfaccia (serve per ottimizzare il processo di marshalling dei dati).

5.2.2.1 Il mapping delle operazioni *oneway*

L'OMG IDL permette di indicare la semantica dell'invocazione di un'operazione nella sua definizione. Questa specifica è fatta tramite l'uso degli attributi delle operazioni. Attualmente, l'unico attributo definito da CORBA è quello *oneway*.

Questo attributo indica che il client, dopo aver fatto la sua invocazione dell'operazione, non ha più nessuna indicazione sull'esito dell'operazione. Cioè, il client fa la sua richiesta, ma non riceve nessuna indicazione da parte del server se l'operazione è andata a buon fine. A causa di questa semantica, i metodi di tipo *oneway* non possono avere parametri in uscita e non possono sollevare eccezioni.

Può sembrare che le operazioni in MIDL e ODL (almeno quelle che non hanno parametri in uscita) abbiano tutte questa semantica. In realtà, ogni metodo COM deve restituire un *HRESULT* e quindi c'è comunque una certa interazione da parte del server. A causa di ciò, non c'è un modo per mappare le operazioni di tipo *oneway* conservandone la semantica esatta, e quindi vengono mappate come se fossero delle normali operazioni che non hanno ovviamente parametri in uscita.

5.2.3 Il mapping degli attributi

L'OMG IDL permette alle interfacce di avere degli attributi. Gli attributi vengono mappati sul linguaggio target essenzialmente con una coppia di funzioni che accedono ad un dato: una per leggerne il valore e l'altra (opzionale) per modificarlo. La definizione dell'attributo deve essere contenuta all'interno della definizione dell'interfaccia e può indicare se il dato è a sola lettura (con la parola chiave *readonly*), o se si può accedere ad esso anche in scrittura. Nell'esempio in OMG IDL che vedremo successivamente, definiamo due attributi: uno all'interno dell'interfaccia *Customer* che è *Profile*, e l'altro, a sola lettura, all'interno dell'interfaccia *Account* che è *Balance*.

Per definire gli attributi in OMG IDL si usa la parola chiave *attribute*. Questi, inoltre, non possono sollevare eccezioni definite dall'utente. Per questo motivo le funzioni che implementano l'attributo possono solamente sollevare eccezioni di sistema (che vengono codificate nel mapping all'interno del valore di ritorno HRESULT in COM).

Veniamo all'esempio:

```
// OMG IDL
typedef ... CustomerId;

struct CustomerData
{
   CustomerId Id;
   string Name;
   string SurName;
};
```

```
#pragma ID::BANK::Account "IDL:BANK/Account:3.1"
interface Account
{
   readonly attribute float Balance;
   float Deposit(in float amount);
   float Withdraw(in float amount);
   float Close();
};

#pragma ID::BANK::Customer "IDL:BANK/Customer:1.2"
interface Customer
{
   attribute CustomerData Profile;
};
```

Quando si passa dall'OMG IDL al MIDL gli attributi vengono trasformati in due funzioni che accedono al dato che hanno lo stesso nome dell'attributo, però con il prefisso *_get_* (per la lettura) e *_set_* (per la scrittura). Nel caso invece dell'ODL, le funzioni conservano il nome esatto dell'attributo, ma hanno il modificatore *[propget]* per quella in lettura e *[propput]* per quella in scrittura.

Gli attributi che sono di tipo *readonly* vengono mappati con la sola funzione in lettura.

Quindi l'esempio precedente viene trasformato in questo codice MIDL:

```
// MIDL
typedef ... CustomerId;
typedef struct
{
   ...
} CustomerData;

[
   object,
   uuid(...),
   pointer_default(unique)
]
interface IAccount : IUnknown
{
   HRESULT _get_Balance([out] float * Balance);
   ...
};

[
   object,
   uuid(...),
   pointer_default(unique)
]
interface ICustomer : IUnknown
```

```
{
   HRESULT _get_Profile([out] CustomerData * Profile);
   HRESULT _set_Profile([in] CustomerData Profile);
};
```

La corrispondente versione ODL è invece:

```
// ODL
typedef ... CustomerId;
typedef struct
{
   ...
} CustomerData;

[
   uuid(...)
]
interface IAccount : IUnknown
{
   [propget] HRESULT Balance([out] float * Balance);
   ...
};

[
   uuid(...)
]
interface ICustomer : IUnknown
{
   [propget] HRESULT Profile([out] CustomerData * Profile);
   [propput] HRESULT Profile([in] CustomerData Profile);
};
```

Inoltre, all'interno della definizione di interfaccia, gli attributi sono ordinati lessicograficamente in base al loro nome e le funzioni *get* precedono quelle *put*. Vedremo, comunque, le regole precise nel prossimo paragrafo.

5.2.4 Il mapping dell'ereditarietà

CORBA e COM hanno un sistema simile per definire le interfacce che ereditano singolarmente; invece i due modelli sono differenti per quello che riguarda l'ereditarietà multipla.

In CORBA, un'interfaccia può ereditare singolarmente o in modo multiplo da altre interfacce. Nei linguaggi target che supportano i puntatori *tipati* si può fare il *casting* del puntatore per accedere all'oggetto (interfaccia) desiderato. Comunque in CORBA non c'è un meccanismo per accedere alle varie interfacce senza usare le relazioni di

ereditarietà. Per quello che riguarda l'implementazione, CORBA permette molti modi per implementare le interfacce, compresa la comodissima *implementation inheritance*.

In COM, almeno nella versione 2.0, le interfacce possono avere (devono avere) solo l'ereditarietà singola. Comunque, al contrario di CORBA, c'è un meccanismo standard tramite il quale un oggetto può avere più interfacce senza usare le relazioni di ereditarietà e i client, con la funzione **QueryInterface**, possono richiederle a tempo di esecuzione. Non c'è però un modo comune per determinare se due riferimenti ad interfaccia puntano alla stessa interfaccia e per enumerare tutte le interfacce supportate da un oggetto.

Un'altra diversità è che in COM gli oggetti hanno un insieme minimo di interfacce che devono supportare e non si può avere una conoscenza diretta di esse se non si ha una **type library**.

COM supporta due tecniche principali per l'implementazione: l'**aggregazione** e il **contenimento** (o **delegazione**). L'ereditarietà dell'implementazione in stile C++ non è possibile.

Mappare le interfacce CORBA in quelle corrispondenti COM è più complicato che mappare le interfacce COM in quelle corrispondenti CORBA, proprio perché CORBA supporta l'ereditarietà multipla e COM no. Vediamo adesso le regole precise che si devono usare quando si passa da un'architettura all'altra.

Come già accennato nel Capitolo 4, se un'interfaccia CORBA eredita singolarmente, viene mappata direttamente con una corrispondente interfaccia COM. Alla base dell'albero dell'ereditarietà degli oggetti CORBA c'è l'interfaccia **Object** che non è supportata in COM (che ha invece **IUnknown**) e deve essere quindi emulata con l'interfaccia **ICORBAObject** (descritta dalla specifica CORBA). Le varie interfacce possono essere richieste in COM con la funzione **IUnknown::QueryInterface**.

Queste regole si applicano per mappare l'ereditarietà da CORBA a COM:

- Ogni interfaccia OMG IDL che non ha un'interfaccia madre si mappa con un'interfaccia MIDL che deriva da IUnknown.
- Ogni interfaccia OMG IDL che eredita da una singola interfaccia madre si mappa con un'interfaccia MIDL che eredita dal mapping dell'interfaccia madre.
- Ogni interfaccia OMG IDL che eredita da più interfacce si mappa con un'interfaccia MIDL che deriva da IUnknown.
- Per ogni interfaccia CORBA, il mapping delle operazioni precede quello degli attributi.
- Il mapping risultante delle operazioni contenute in un'interfaccia viene ordinato in base al nome delle operazioni. L'ordine è lessicografico.
- Il mapping risultante degli attributi contenuti all'interno di un'interfaccia viene ordinato in base al nome degli attributi. L'ordine è lessicografico. Se un attributo non è a sola lettura, il metodo *get* precede il metodo *set*.

Vediamo un esempio di mapping (che è simile a quello del Capitolo 4). L'albero dell'ereditarietà è schematizzato nella Figura 5.3.

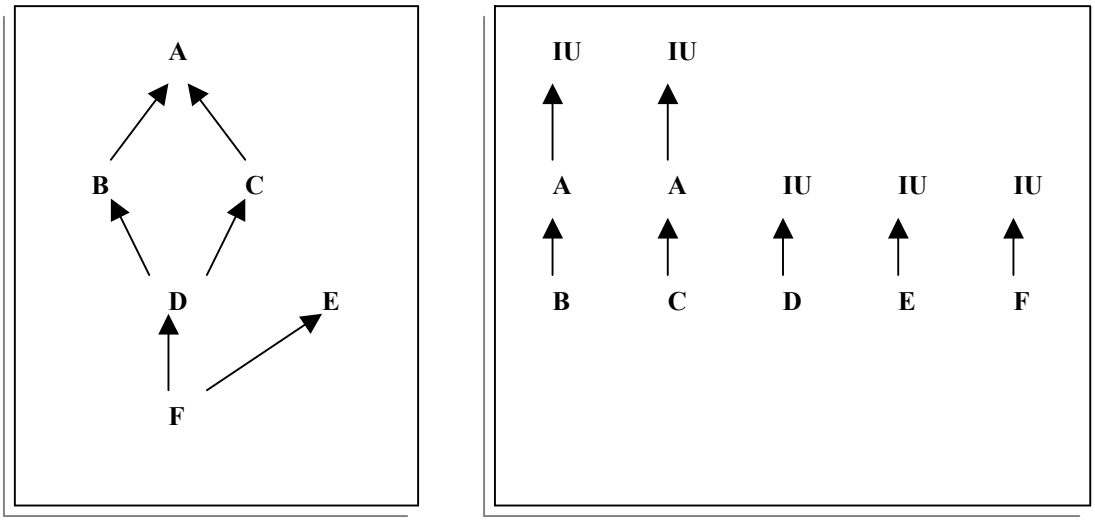


Figura 5.3 Schema dell'ereditarietà in CORBA (a) e il corrispondente mapping in COM (b).

```
// OMG IDL
interface A
{
   void opA();
   attribute long val;
};
interface B : A
{
   void opB();
};
interface C : A
{
   void opC();
};
interface D : B, C
{
   void opD();
};
interface E
{
   void opE();
};
interface F : D, E
{
```

```
   void opF();
};
```

Il corrispondente mapping in MIDL e rappresentato da questo frammento di codice:

```
// MIDL
[
   object, uuid(...), pointer_default(unique)
]
interface IA : IUnknown
{
   HRESULT opA();
   HRESULT _get_val([out] long * val);
   HRESULT _set_val([in] long val);
};
[
   object, uuid(...), pointer_default(unique)
]
interface IB : IA
{
   HRESULT opB();
};
[
   object, uuid(...), pointer_default(unique)
]
interface IC : IA
{
   HRESULT opC();
};
[
   object, uuid(...), pointer_default(unique)
]
interface ID : IUnknown
{
   HRESULT opD();
};
[
   object, uuid(...), pointer_default(unique)
]
interface IE : IUnknown
{
   HRESULT opE();
};
[
   object, uuid(...), pointer_default(unique)
]
interface IF : IUnknown
{
   HRESULT opF();
};
```

In COM le interfacce si associano alle implementazioni degli oggetti nel blocco **coclass**. Se, ad esempio, vogliamo che la classe *D* implementi le interfacce *IB* e *IC*, oltre alla ovvia *ID*, (come specificato nello schema CORBA) dobbiamo scrivere qualcosa del genere:

```
// MIDL
[
   uuid(...)
]
coclass D
{
   [default] interface ID;
   interface IB;
   interface IC;
};
```

Nel passaggio dalle definizioni delle interfacce OMG IDL a quelle corrispondenti MIDL, si aggiunge al nome delle interfacce la lettera "I" per indicare che il nome rappresenta un'interfaccia. Questo rende il mapping più naturale ed agevole per i programmatori COM dato che la Microsoft stessa suggerisce di usare questa convenzione.

5.3 Il mapping delle eccezioni

Il sistema CORBA usa il concetto di eccezione per restituire informazioni sugli errori. Inoltre ci possono essere delle ulteriori informazioni, che accompagnano l'eccezione, che consistono in una forma specializzata di record. All'interno del record ci può essere qualsiasi informazione il cui tipo è di base o complesso. Le eccezioni possono essere di due tipi: di sistema, e definite dall'utente.

COM, invece, restituisce le informazioni sugli errori ai client tramite il valore di ritorno *HRESULT*. Se vale zero indica che l'operazione è andata a buon fine. L'*HRESULT* può essere convertito in un codice chiamato **SCODE** (sulla piattaforma Win32 i due valori coincidono) che può essere esaminato per vedere il tipo dell'errore che si è eventualmente verificato. Il codice di errore è composto da due parti, una di 13 bit (quelli più significativi), e una di 16 (quelli meno significativi). Sulla oramai obsoleta piattaforma Win16 la prima parte è composta di soli 4 bit.

A differenza di CORBA, COM non fornisce un metodo standard per restituire ai client le eccezioni definite dall'utente. Gli errori che un'invocazione di funzione può restituire sono definiti staticamente quando la si definisce e non possono essere cambiati. Inoltre non c'è un modo per i client per sapere a tempo di esecuzione quali errori può ricevere da un'interfaccia.

Poiché il modello delle eccezioni CORBA è significativamente più ricco di quello COM, nella fase del mapping si deve aggiungere un ulteriore protocollo a COM. Nel fare ciò, si deve comunque mantenere la compatibilità verso il basso e non si devono apportare dei cambiamenti al modello.

In conclusione, per restituire le eccezioni definite dall'utente ai client, si aggiunge alle operazioni un ulteriore parametro opzionale che serve appunto per restituire l'eccezione.

5.3.1 Il mapping delle eccezioni di sistema

Le eccezioni di sistema sono standard e sono definite dalla specifica CORBA. Sono usate automaticamente dall'ORB quando si verificano condizioni anomale. Possono essere restituite come risultato dell'invocazione di qualsiasi operazione, indipendentemente dall'interfaccia o dall'operazione invocata.

Ci sono due aspetti che si devono considerare quando si mappano le eccezioni di sistema. Uno è quello di generare un appropriato codice *HRESULT* da far restituire alla View, l'altro è convertire le informazioni sull'errore in un oggetto **OLE Error** e restituirlo al client.

La Figura 5.4 mostra gli *HRESULT* che una COM View deve restituire quando si verificano delle eccezioni di sistema in CORBA. A queste viene assegnato un identificatore numerico a 16 bit, a partire dal valore esadecimale 0x200 che viene usato come codice d'errore e viene memorizzato nella parte bassa dell'*HRESULT*. I bit 12 e 13 contengono una maschera che indica lo stato di completamento della richiesta CORBA. Il valore 00 indica che l'operazione non è stata completata, mentre 01 indica che è andata a buon fine. Il valore 10 è un po' particolare, perché indica che l'operazione potrebbe essere stata completata con successo. Infine il valore 11 non è usato.

Eccezioni di sistema	Valore HRESULT
ITF_E_UNKNOWN_NO	0x40200
ITF_E_UNKNOWN_YES	0x41200
ITF_E_UNKNOWN_MAYBE	0x42200
ITF_E_BAD_PARAM_NO	0x40201
ITF_E_BAD_PARAM_YES	0x41201
ITF_E_BAD_PARAM_MAYBE	0x42201
ITF_E_MEMORY_NO	0x40202

ITF_E_MEMORY_YES	0x41202
ITF_E_MEMORY_MAYBE	0x42202
ITF_E_IMP_LIMIT_NO	0x40203
ITF_E_IMP_LIMIT_YES	0x41203
ITF_E_IMP_LIMIT_MAYBE	0x42203
ITF_E_COMM_FAILURE_NO	0x40204
ITF_E_COMM_FAILURE_YES	0x41204
ITF_E_COMM_FAILURE_MAYBE	0x42204
ITF_E_INV_OBJREF_NO	0x40205
ITF_E_INV_OBJREF_YES	0x41205
ITF_E_INV_OBJREF_MAYBE	0x42205
ITF_E_NO_PERMISSION_NO	0x40206
ITF_E_NO_PERMISSION_YES	0x41206
ITF_E_NO_PERMISSION_MAYBE	0x42206
ITF_E_INTERNAL_NO	0x40207
ITF_E_INTERNAL_YES	0x41207
ITF_E_INTERNAL_MAYBE	0x42207
ITF_E_MARSHAL_NO	0x40208
ITF_E_MARSHAL_YES	0x41208

ITF_E_MARSHAL_MAYBE	0x42208
ITF_E_INITIALIZE_NO	0x40209
ITF_E_INITIALIZE_YES	0x41209
ITF_E_INITIALIZE_MAYBE	0x42209
ITF_E_NO_IMPLEMENT_NO	0x4020A
ITF_E_NO_IMPLEMENT_YES	0x4120A
ITF_E_NO_IMPLEMENT_MAYBE	0x4220A

ITF_E_BAD_TYPECODE_NO	0x4020B
ITF_E_BAD_TYPECODE_YES	0x4120B
ITF_E_BAD_TYPECODE_MAYBE	0x4220B
ITF_E_BAD_OPERATION_NO	0x4020C
ITF_E_BAD_OPERATION_YES	0x4120C
ITF_E_BAD_OPERATION_MAYBE	0x4220C
ITF_E_NO_RESOURCES_NO	0x4020D
ITF_E_NO_RESOURCES_YES	0x4120D
ITF_E_NO_RESOURCES_MAYBE	0x4220D
ITF_E_NO_RESPONSE_NO	0x4020E
ITF_E_NO_RESPONSE_YES	0x4120E
ITF_E_NO_RESPONSE_MAYBE	0x4220E
ITF_E_PERSIST_STORE_NO	0x4020F
ITF_E_PERSIST_STORE_YES	0x4120F
ITF_E_PERSIST_STORE_MAYBE	0x4220F
ITF_E_BAD_INV_ORDER_NO	0x40210
ITF_E_BAD_INV_ORDER_YES	0x41210
ITF_E_BAD_INV_ORDER_MAYBE	0x42210
ITF_E_TRANSIENT_NO	0x40211
ITF_E_TRANSIENT_YES	0x41211

ITF_E_TRANSIENT_MAYBE	0x42211
ITF_E_FREE_MEM_NO	0x40212
ITF_E_FREE_MEM_YES	0x41212
ITF_E_FREE_MEM_MAYBE	0x42212
ITF_E_INV_IDENT_NO	0x40213
ITF_E_INV_IDENT_YES	0x41213
ITF_E_INV_IDENT_MAYBE	0x42213

ITF_E_INV_FLAG_NO	0x40214
ITF_E_INV_FLAG_YES	0x41214
ITF_E_INV_FLAG_MAYBE	0x42214
ITF_E_INTF_REPOS_NO	0x40215
ITF_E_INTF_REPOS_YES	0x41215
ITF_E_INTF_REPOS_MAYBE	0x42215
ITF_E_BAD_CONTEXT_NO	0x40216
ITF_E_BAD_CONTEXT_YES	0x41216
ITF_E_BAD_CONTEXT_MAYBE	0x42216
ITF_E_OBJ_ADAPTER_NO	0x40217
ITF_E_OBJ_ADAPTER_YES	0x41217
ITF_E_OBJ_ADAPTER_MAYBE	0x42217
ITF_E_DATA_CONVERSION_NO	0x40218
ITF_E_DATA_CONVERSION_YES	0x41218
ITF_E_DATA_CONVERSION_MAYBE	0x42218

Figura 5.4 Mapping tra eccezioni di sistema e HRESULT.

Non è possibile inserire l'ulteriore codice delle eccezioni di sistema (**codice minore**) e il RepositoryId all'interno dell'*HRESULT*. Quindi si può usare un oggetto del tipo OLE Error per convertire e restituire questi dati. Comunque, se si usa questo metodo, lo si deve fare seguendo la seguente specifica:

- La COM View deve implementare l'interfaccia COM standard **ISupportErrorInfo** in modo tale da poter poi restituire ai client l'oggetto OLE Error.
- La COM View deve chiamare la funzione **SerErrorInfo** con il parametro **IErrorInfo** impostato a *NULL* quando l'invocazione dell'operazione CORBA è andata a buon fine e non ha sollevato eccezioni. In questo modo si assicura che l'oggetto OLE Error sia completamente distrutto.

Le proprietà dell'oggetto OLE Error devono essere *settati* in base a quanto specificato nella Figura 5.5.

Proprietà	Descrizione
bstrSource	<nome interfaccia>.<nome operazione> Dove il nome dell'interfaccia e quello dell'operazione sono quelli dell'interfaccia CORBA che la View rappresenta.
bstrDescription	Eccezione di sistema di CORBA: [<RepositoryId dell'eccezione>] [<codice minore>] [<stato di completamento>] Dove il RepositoryId e il codice minore sono quelli dell'eccezione di sistema di CORBA. Lo stato di completamento può essere: "YES", "NO" oppure "MAYBE", in base al valore dell'eccezione di sistema. Gli spazi e le parentesi quadre sono caratteri validi e devono essere inclusi nella stringa.
bstrHelpFile	Non specificato
dwHelpContext	Non specificato
GUID	L'IID dell'interfaccia della COM View

Figura 5.5 Uso dell'oggetto OLE Error per le eccezioni di sistema di CORBA.

Una COM View che supporta gli oggetti OLE Error per le eccezioni di sistema CORBA, dovrebbe avere un codice (in C++) simile a questo:

```
// COM View
SetErrorInfo(0L, NULL); // Inizializza l'oggetto OLE Error
try
{
   // Invocazione dell'operazione CORBA
}
catch(...)
{
   ...
   CreateErrorInfo(&pICreateErrorInfo);
   pICreateErrorInfo->SetSource(...);
   pICreateErrorInfo->SetDescription(...);
   pICreateErrorInfo->SetGUID(...);
   pICreateErrorInfo->QueryInterface(IID_IErrorInfo,
                                     &pIErrorInfo);
   pICreateErrorInfo->SetErrorInfo(0L, pIErrorInfo);
   pIErrorInfo->Release();
   pICreateErrorInfo->Release();
   ...
};
```

Invece il codice di un client di una COM View per accedere alle informazioni presenti nell'oggetto OLE Error dovrebbe essere simile a questo:

```
// Client di una COM View

// Dopo aver ottenuto un puntatore ad un'interfaccia
// della COM View, il client dovrebbe fare la seguente
// cosa una sola volta:

pIMyMappedInterface->QueryInterface(IID_ISupportErrorInfo,
                                    &pISupportErrorInfo);
HRESULT hr = pISupportErrorInfo->InterfaceSupportsErrorInfo
   (IID_MyMappedInterface);
BOOL bSupportsErrorInfo = (hr = NOERROR ? TRUE : FALSE);
...
// Chiama l'operazione COM
HRESULT hrOperation = pIMyMappedInterface->...

if (bSupportsErrorInfo)
{
   HRESULT hr = GetErrorInfo(0L, &pIErrorInfo);
   // S_FALSE significa che i dati sull'errore
   // non sono disponibili, NO_ERROR indica
   // invece che ci sono
   if (hr = NO_ERROR)
   {
      pIErrorInfo->GetSource(...);
      pIErrorInfo->GetDescription(...);
      pIErrorInfo->GetGUID(...);
      // Il codice di completamento è contenuto
      // in hrOperation
   }
};
```

Il client COM può usare il meccanismo di gestione delle eccezioni del C++ per evitare di fare un controllo esplicito su ogni chiamata delle operazioni della COM View.

5.3.2 Il mapping delle eccezioni definite dall'utente

Le eccezioni definite dall'utente sono scritte in OMG IDL e sono usate dai metodi di un oggetto server per restituire gli errori specifici di un operazione. La definizione di un'eccezione definita dall'utente è identificata all'interno di un file dalla parola chiave *exception*. Il suo corpo si definisce con la stessa sintassi che si usa per descrivere le strutture.

Quando le eccezioni definite dall'utente vengono mappate in COM, una struttura è usata per descrivere le sue informazioni. Essa contiene dei membri che indicano il tipo dell'eccezione CORBA, il suo identificatore nella forma di Repository Id ed il

puntatore all'interfaccia. Il nome della struttura è costruito in base al nome del modulo in cui essa è contenuta (se esiste), al nome dell'interfaccia che la usa e dalla parola *Exceptions*. Vediamo un esempio chiarificatore:

```
// MIDL o ODL
typedef enum { NO_EXCEPTION, USER_EXCEPTION }
   ExceptionType;

typedef struct
{
   ExceptionType type;
   LPTSTR repositoryId;
   <nome modulo><nome interfaccia>UserException
      *...piUserException;
} <nome modulo><nome interfaccia>Exceptions;
```

La struttura dell'eccezione è specificata come parametro di uscita e appare alla fine della lista dei parametri di un'operazione (risultante dal mapping) che solleva un'eccezione definita dall'utente. È sempre passata come riferimento indiretto. A causa delle regole della gestione della memoria in COM, passando la struttura dell'eccezione come riferimento indiretto in un parametro di uscita, permette al chiamante di trattarla come un parametro opzionale. Il seguente esempio illustra questo punto:

```
// MIDL
[
   object,
   uuid(...),
   pointer_default(unique)
]
interface IAccount
{
   HRESULT Withdraw([in] float Amount,
                    [out] float pNewBalance,
                    [out] BankExceptions ** ppException);
};
```

Il chiamante può indicare che non vuole ricevere le informazioni dell'eccezione, se si verifica, specificando il valore *NULL* per il parametro dell'eccezione. Se al contrario vuole ricevere le informazioni, deve passare l'indirizzo di un puntatore a una zona di memoria in cui saranno memorizzate. La gestione della memoria in COM suppone che sia il client a liberare questa memoria quando non ne ha più bisogno.

Se il chiamante fornisce un puntatore non nullo come parametro dell'eccezione e il chiamato restituisce un'eccezione, è quest'ultimo che si deve preoccupare dell'allocazione della memoria che conterrà le eventuali informazioni. Se l'eccezione non si verifica, il chiamato non usa il parametro.

Se non si verifica nessuna eccezione CORBA, si deve restituire il valore *S_OK* come *HRESULT* al chiamato per indicare che l'operazione ha avuto successo. Il valore dell'*HRESULT* restituito al chiamato, quando si è verificata un'eccezione CORBA, dipende dal tipo dell'eccezione e se è stata specificata una struttura dal chiamante. Vediamo un esempio di definizione di eccezioni in OMG IDL e il loro relativo mapping:

```
// OMG IDL
module BANK
{
   ...
   exception InsufFunds { float balance };
   exception InvalidAmount { float amount };
   ...
   interface Account
   {
      exception NotAuthorized {};
      float Deposit(in float Amount)
         raises(InvalidAmount);
      float Withdraw(in float Amount)
         raises(InvalidAmount, NotAuthorized);
   };
};
```

Questo è il relativo mapping in MIDL e ODL:

```
// MIDL e ODL
struct BANKInsufFunds
{
   float balance;
};
struct BANKInvalidAmount
{
   float amount;
};
struct BANKAccountNotAuthorized
{
};
[
   object,
   uuid(...),
   pointer_default(unique)
]
interface IBANKAccountUserException : IUnknown
{
   HRESULT _get_InsufFunds([out] BankInsufFunds
      * exceptionBody);
   HRESULT _get_InvalidAmount([out] BankInvalidAmount
      * exceptionBody);
   HRESULT _get_NotAuthorized([out] BankAccountNotAuthorized
```

```
      * exceptionBody);
};

typedef struct
{
   ExceptionType type;
   LPTSTR repositoryId;
   IBANKAccountUserExceptions * piUserException;
} BANKAccountExceptions;
```

Le eccezioni definite dall'utente sono mappate in COM con un'interfaccia (**Exception interface**) ed una struttura (**Exception structure**) che descrive (definisce) le informazioni che devono essere restituite dall'eccezione. Si definisce un'interfaccia COM per ogni interfaccia CORBA che restituisce eccezioni definite dall'utente (oltre a quella relativa al mapping vero e proprio). Il nome dell'interfaccia è costruito in base allo **scope** (tiene conto di tutti i moduli annidati) dell'interfaccia CORBA. La struttura si definisce invece per ogni eccezione definita dall'utente e contiene il corpo delle informazioni che saranno restituite con l'eccezione. Il nome di questa struttura è costruito in base alle normali regole che definiscono il mapping delle strutture.

Ogni eccezione definita dall'utente che può essere sollevata da un'operazione viene mappata con un'operazione della Exception interface. Il nome dell'operazione è costruito col nome dell'eccezione preceduto dalla stringa *"_get_"*. Ogni operazione ha un parametro di uscita che punta al corpo delle informazioni restituite dall'eccezione. Il tipo del parametro di uscita è la Exception structure.

L'operazione restituisce un HRESULT. Se viene sollevata un'eccezione CORBA definita dall'utente, verrà restituito nell'*HRESULT* il valore *E_FAIL*.

Se il chiamante specifica un valore non nullo (diverso da *NULL*) per il parametro della Exception structure, il chiamato deve allocare la memoria necessaria per la struttura e la deve riempire come specificato nella Figura 5.6.

Quando si verifica un errore di conversione dei dati tra i due sistemi (durante la chiamata da un client COM ad un server CORBA), un HRESULT con il codice E_DATA_CONVERSION e il valore FACILITY_NULL dell'ulteriore parametro (**Facility value**) devono essere restituiti al client.

Membro	**Descrizione**
Type	Indica il tipo dell'eccezione CORBA che è stata sollevata.
RepositoryId	Indica il RepositoryId dell'eccezione.
PiUserException	Punta all'interfaccia dalla quale è possibile ottenere informazioni sull'eccezione.

Figura 5.6 Descrizione dei campi della **Exception structure**.

Capitolo 6

Implementazione

La specifica dell'**OMG Group** relativa all'interazione tra COM e CORBA è stata implementata dall'autore di questo documento apportando delle modifiche al compilatore IDL della distribuzione **Mico** di CORBA ed è stato ivi incluso a partire dalla versione 2.2.7. Per maggiori approfondimenti, è possibile consultare il sito *http://www.mico.org*.

6.1 La distribuzione Mico

L'acronimo **MICO** si espande in **M**ico **I**s **CO**RBA. L'intenzione di questo progetto è di fornire un'implementazione dello standard CORBA 2.2 completamente libera e totalmente compatibile. Mico è diventato molto popolare come progetto **OpenSource** ed è largamente usato per differenti scopi. I sorgenti di Mico sono distribuiti sotto la licenza GNU.

Lo scopo dei progettisti di Mico è quello di mantenerlo compatibile con l'ultimo standard di CORBA rilasciato dal Gruppo OMG. Tutto quello che è implementato fa parte delle specifica CORBA 2.2, incluse, ma non limitate, alle seguenti caratteristiche:

- **Dynamic Invocation Interface** (**DII**).
- **Dynamic Skeleton Interface** (**DSI**).
- Mapping da IDL a C++.
- **Interface Repository** (**IR**).
- **Browser** grafico dell'Interface Repository che permette di invocare un metodo arbitrario su una qualsiasi interfaccia.
- **IIOP** come protocollo nativo.
- **IIOP** su **SSL.**
- Design modulare dell'ORB: dei nuovi protocolli di comunicazione e dei nuovi Object Adapter possono essere facilmente collegati all'ORB, anche a tempo di esecuzione, usando dei moduli caricabili.

- Supporto delle invocazioni annidate dei metodi.
- **Interceptor**.
- Supporto del tipo *any*: offre un'interfaccia per inserire ed estrarre dei tipi di dato che non sono conosciuti a tempo di compilazione.
- Supporto dei **TypeCode**: servono per descrivere a tempo di esecuzione tutti i componenti di un'interfaccia.
- Supporto dei tipi di dato ricorsivi.
- Implementazione completa del BOA, con tutte le politiche di attivazione, la migrazione e la persistenza degli oggetti, e l'Implementation Repository.
- Il BOA può caricare le implementazioni degli oggetti all'interno dei client usando i moduli.
- **Portable Object Adapter** (**POA**).
- Supporto delle applicazioni X11 (Xt e Qt).
- **Interoperable Naming Service** (**INS**).
- **Event Service.**
- **Relationship Service.**
- **Property Service.**
- **Trading Service.**
- Supporto per il tipo *DynAny*: corrisponde al tipo *any* dinamico.

6.2 Il compilatore IDL

La distribuzione Mico viene rilasciata sotto forma di sorgenti compressi con il formato *tar.gz* oppure *zip*. Dopo averli scompattati si ottiene una *directory*, chiamata *"Mico"*, che contiene tutti i sorgenti e la documentazione sull'installazione e l'uso per un totale di circa 8 MB.

Cercando tra le *sottodirectory* di Mico, troviamo la cartella *"IDL"* che contiene tutti i sorgenti necessari per generare il compilatore IDL. Oltre ai file relativi allo **scanner** e al **parser**, ne troviamo altri che definiscono le classi che servono per gestire le opzioni da riga di comando del compilatore. Cerchiamo adesso di dare un'idea sul funzionamento generale del programma.

La classe **IDLParam**, definita nel file *Param.h*, serve per tenere traccia delle opzioni da riga di comando che sono state specificate al momento dell'esecuzione del compilatore. Contiene dei dati membro che corrispondono ognuno ad un'opzione. Ad esempio c'è la variabile:

```
bool codegen_cpp;
```

che indica se è stato specificato al compilatore di generare il codice per lo **stub** e lo **skeleton** in C++, oppure c'è la variabile:

```
string name;
```

che memorizza il nome specificato dall'utente per il file di *output* (altrimenti si usa il nome di *default*). La classe controlla inoltre le opzioni che sono mutuamente esclusive e quelle che devono essere specificate insieme.

Nel *main*, con degli *if* annidati si testa il valore di queste variabili e si eseguono le operazioni corrispondenti. Ad esempio, con le istruzioni:

```
if (params.codegen_cpp) {
   CodeGen *gen = new CodeGenCPP(db, params);
   gen->emit(name);
   delete gen;
}
```

si testa il *flag* per la generazione del codice C++ e in caso positivo si genera il codice corrispondente.

Come ho detto in precedenza, ogni opzione da riga di comando del compilatore viene gestita da una classe e **CodeGenCPP** si occupa appunto della generazione del codice C++. Al suo costruttore viene passato il *database*, generato dal **parser**, che contiene le informazioni estratte dal file IDL e gli ulteriori parametri che influenzano la generazione dello **stub** e dello **skeleton**. La generazione vera e propria viene fatta dalla funzione **emit** a cui viene passato il nome del file di *output*. Tutte le altre opzioni vengono gestite più o meno allo stesso modo.

6.3 Schema dell'implementazione

Per motivi di omogeneità, ho cercato di implementare il traduttore da OMG IDL a Microsoft IDL mantenendo lo stesso approccio usato per le altre opzioni da riga di comando. Ho modificato i file originali *Makefile*, *param.h*, *param.cc*, *main.cc* e ho creato i file *codegen-midl.h* e *codegen-midl.cc*. Quest'ultimi definiscono la classe **CodeGenMIDL** che costituisce il traduttore vero o proprio.

Al *Makefile* ho semplicemente aggiunto il comando per far compilare e *linkare* anche i due nuovi file *codegen-midl.h* e *codegen-midl.cc*.

Fra i comandi che si possono impartire al compilatore ho aggiunto le opzioni *--codegen-midl* e *--no-codegen-midl* che indicano rispettivamente se generare o meno il codice Microsoft IDL. Vediamo come vengono gestite.

Al file *param.h* ho aggiunto la variabile membro:

```
   bool codegen_midl;
```

che serve per indicare se è stata specificata o meno l'opzione per la generazione del codice MIDL.

Nel file *param.cc*, invece, ho aggiunto le istruzioni:

```
...
opts["--codegen-midl"]    = "";
opts["--no-codegen-midl"] = "";
...
} else if (arg == "--codegen-midl") {
   codegen_midl = true;
} else if (arg == "--no-codegen-midl") {
   codegen_midl = false;
...
```

che servono per il *parsing* dei parametri ed ho fatto delle ulteriori modifiche per settare i valori di default e per controllare la consistenza delle opzioni (ad esempio non si possono specificare le opzioni *--codegen-idl* e *--codegen-midl* usando un solo parametro *--name*).

Al file *main.cc* ho aggiunto le istruzioni:

```
if (params.codegen_midl) {
   CodeGen *gen = new CodeGenMIDL(db, params.emit_repoids);
   gen->emit(name);
   delete gen;
```

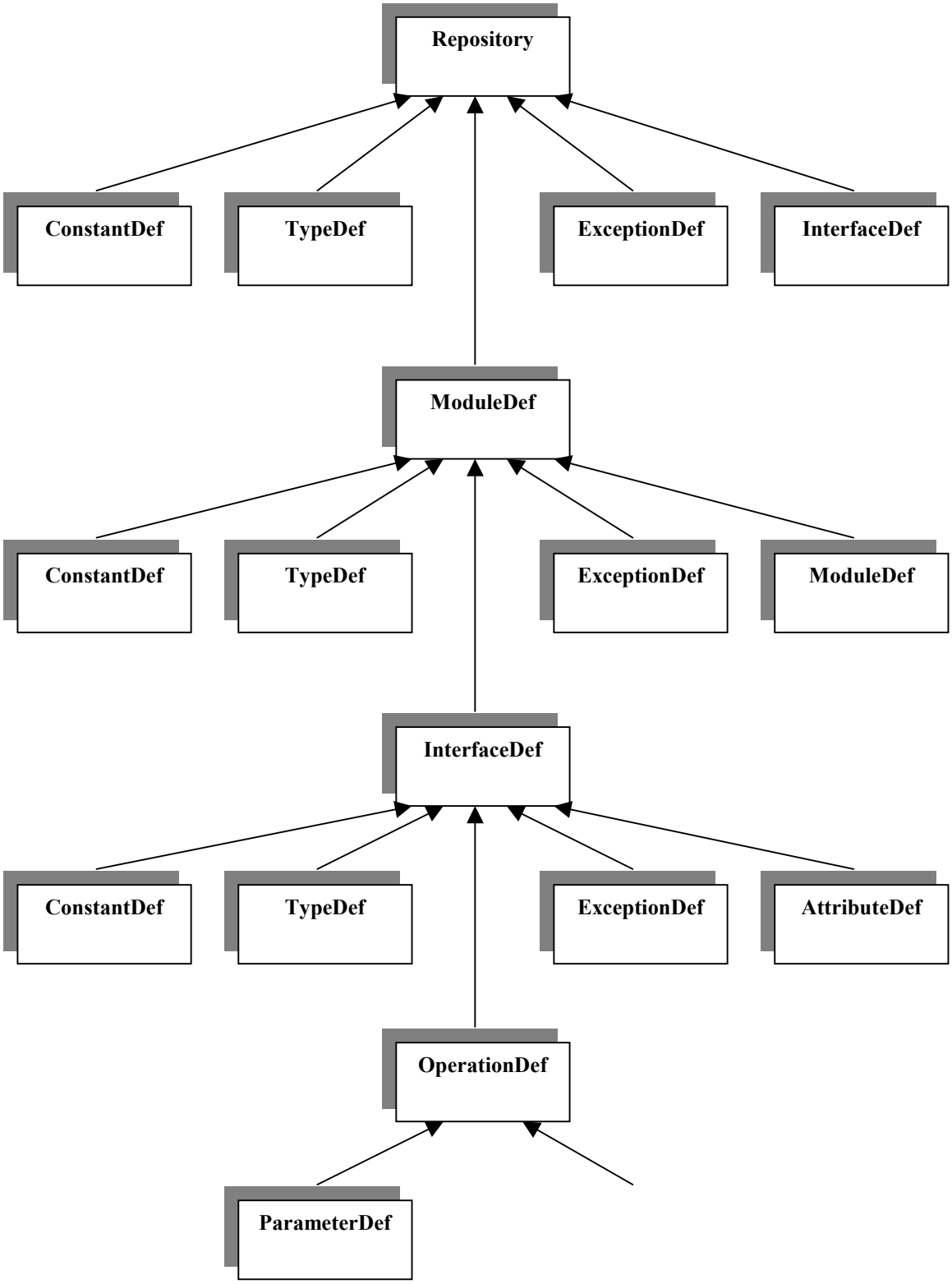
la cui spiegazione e molto simile a quella vista in precedenza. Anche qui si testa il *flag* per la generazione del file MIDL e, in caso si positivo, si istanzia la classe **CodeGenMIDL** che con la funzione **emit** si occuperà della traduzione. Prima di descrivere nei dettagli la classe **CodeGenMIDL**, è necessario spiegare il funzionamento del *parser* IDL.

Come abbiamo visto nel Capitolo 3, la specifica CORBA descrive degli oggetti (sono in realtà delle classi astratte) che servono per rappresentare il contenuto dell'Interface Repository. Questi sono:

- **ModuleDef** che descrive un gruppo logico di interfacce, cioè un modulo.
- **InterfaceDef** che descrive un'interfaccia.
- **OperationDef** che descrive un'operazione di un'interfaccia.
- **ParameterDef** che descrive un parametro di un metodo.
- **AttributeDef** che descrive un attributo di un'interfaccia.
- **ConstantDef** che descrive una costante con nome.
- **ExceptionDef** che descrive un'eccezione che può essere sollevata da un'operazione.
- **TypeDef** che descrive un tipo.

Inoltre, la specifica CORBA definisce anche gli oggetti **IRObject**, **Repository**, **Contained** e **Container**. Il primo mette a disposizione un metodo per identificare il tipo di un oggetto e un metodo per distruggerlo. Il secondo fornisce i metodi comuni a tutti gli oggetti dell'Interface Repository. Il terzo fornisce le funzioni per spostare gli oggetti da un contenitore (**Container**) ad un altro e per descrivere il suo contenuto. Il quarto fornisce i metodi per *navigare* all'interno di un oggetto contenitore.

La Figura 3.3 del Capitolo 3 mostra la gerarchia di ereditarietà di questi oggetti, mentre la loro gerarchia di contenimento è mostrata nella Figura 6.1.



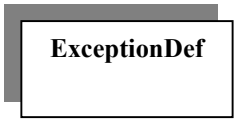


Figura 6.1 Schema della gerarchia di contenimento degli oggetti dell'Interface Repository.

I progettisti di Mico hanno implementato queste classi e le hanno usate sia per gli oggetti dell'Interface Repository, che per il **parser** IDL. Quest'ultimo infatti, quando analizza un file IDL, memorizza le definizioni incontrate in istanze delle classi viste prima e le raggruppa in un *database* che è definito nei file *db.h* e *db.cc*. La classe **CodeGenMIDL** prende come parametro il contenuto del *database* e genera il file MIDL corrispondente.

Ad una visione superficiale l'OMG IDL ed il Microsoft IDL sembrano molto simili, ma in realtà ci sono molti particolari che li rendono abbastanza diversi e che, nel mio caso, hanno notevolmente aumentato il codice da scrivere. Per avere un'idea di ciò, basta pensare al fatto che il MIDL non ha i moduli e che i suoi metodi possono (devono) restituire solamente il tipo *HRESULT*.

Ecco un frammento della definizione della classe **CodeGenMIDL** in cui sono elencati i suoi elementi più importanti:

```
class CodeGenMIDL : public CodeGen
{
private:
   ...
   vector<CORBA::InterfaceDef_var> _coclasslist;
   CORBA::Container_var _current_scope;
   ...
   void emitMIDL();

   void emitMIDLInterface(CORBA::InterfaceDef_ptr in);
   void emitMIDLStruct(CORBA::StructDef_ptr s,
                       bool emit_semicolon = true);
   void emitMIDLUnion(CORBA::UnionDef_ptr u,
                      bool emit_semicolon = true);
   void emitMIDLConstant(CORBA::ConstantDef_ptr co);
   void emitMIDLEnum(CORBA::EnumDef_ptr e,
                     bool emit_semicolon = true);
   void emitMIDLAlias(CORBA::AliasDef_ptr a);
   void emitMIDLAttribute(CORBA::AttributeDef_ptr attr);
   void emitMIDLOperation(CORBA::OperationDef_ptr op);

   void emitPrototypes(CORBA::Container_ptr in);
   void emitForwardDcl(CORBA::Container_ptr in);
```

```
    void emitLocalDecls(CORBA::Container_ptr in);
    void emitLocal(CORBA::Contained_ptr in,
                   bool emit_semicolon = true);

    void emit_midl_type_name(CORBA::IDLType_ptr t);
    void emit_midl_type(CORBA::IDLType_ptr t);
    bool emit_base_type(CORBA::IDLType_ptr t);

    void emit_sequence(CORBA::IDLType_ptr t);
    void emit_array(CORBA::IDLType_ptr t);
    void emit_array_suffix(CORBA::IDLType_ptr t);

    bool Is_Alias_Interface(CORBA::IDLType_ptr t);
    bool Is_Interface(CORBA::IDLType_ptr t);

    void emitHeader();
    void emitInterfaceHeader();
    void emitTypeLib(string &fnbase);
    void insert_guid();

    // Generano solamente un messaggio d'errore
    void emitMIDLException(CORBA::ExceptionDef_ptr e);
    void emitMIDLNative(CORBA::NativeDef_ptr n);
    void emitMIDLValue(CORBA::ValueDef_ptr v);
    void emitMIDLValueBox(CORBA::ValueBoxDef_ptr v);
    void emitValueMember(CORBA::ValueMemberDef_ptr vmd);

    // Altre funzioni helper
    ...

public:
    CodeGenMIDL(DB &db, bool emit_repoids);
    void emit(string &fnbase);
    ...
};
```

Vediamo adesso una sommaria descrizione dei principali metodi:

- **emitMIDLStruct**, **emitMIDLUnion**, **emitMIDLConstant** ed **emitMIDLEnum** sono state le funzioni più semplici da implementare e si occupano rispettivamente della traduzione delle strutture, unioni, costanti ed enumerati. In questo caso infatti, esiste un *mapping* esatto tra OMG IDL e Microsoft IDL.
- **emitMIDLInterface** si occupa della traduzione delle interfacce. Inoltre, memorizza nella variabile *_coclasslist*, che verrà usata nella generazione della **Type Library**, la lista delle eventuali interfacce di base da cui si eredita. Siccome il MIDL non supporta l'ereditarietà multipla, le interfacce di base vengono inserite nella **Type Library**. Vediamo un esempio chiarificatore. Supponiamo di avere questa dichiarazione OMG IDL in cui definiamo un'interfaccia che deriva da altre tre interfacce:

```
// OMG IDL

interface Derived : Base1, Base2, Base3
{
   ...
};
```

Dalla parte MIDL otteniamo:

```
// Microsoft IDL

...
interface IDerived : IUnknown
{
   ...
};
...
library DerivedLib
   ...
   coclass Derived
   {
      [default] interface IDerived;
      interface IBase1;
      interface IBase2;
      interface IBase3;
   };
   ...
};
```

Nella dichiarazione dell'interfaccia abbiamo perso le informazioni sulle interfacce di base, ma quando generiamo la **Type Library** specifichiamo che la classe che implementerà l'interfaccia *IDerived* dovrà implementare anche le interfacce *IBase1*, *IBase2*, e *IBase3*.

- **emitMIDLAlias** si occupa della traduzione dei *typedef*. Non c'è un *mapping* esatto in questo caso tra OMG IDL e MIDL. Infatti, in quest'ultimo non si possono definire degli *alias* per le interfacce, ma solamente per i puntatori ad esse. I due metodi **Is_Interface** ed **Is_Alias_Interface** servono appunto per intercettare questo caso e per trasformarlo in un *alias* ad un puntatore.
- **emitPrototypes** ha il compito di estrarre dalle interfacce sia la sequenza di attributi che quella delle operazioni, di ordinarle in base al loro nome (come previsto dalla specifica CORBA relativa al *mapping*) e di chiamare per ogni elemento le funzioni **emitAttribute** ed **emitOperation**. La prima, per ogni attributo genera due prototipi di funzioni, uno per leggere il valore (*[propget]*) e una per scriverlo (*[propput]*). La seconda genera i prototipi delle operazioni, facendo attenzione a riordinare i modificatori *in* e *out* dell'OMG IDL ed a trasformare i valori di ritorno negli attributi *[out, retval]* del MIDL.
- **emitLocalDecls** viene chiamata all'interno di **emitMIDLInterface** e si occupa della generazione del codice per quelle strutture dati (come ad esempio le strutture

e le unioni) che sono locali ad un'interfaccia. Prende come parametro una sequenza di dichiarazioni (*CORBA::ContainedSeq_var*) e per ogni elemento chiama la funzione **emitLocal**. Quest'ultima, con uno *switch* fa un test sul tipo dell'elemento e, in base a questo, chiama le funzioni corrispondenti che si occupano della generazione del codice vero e proprio. Ecco un suo estratto che chiarirà meglio il discorso:

```
void CodeGenMIDL::emitLocal(CORBA::Contained_ptr con,
                            bool emit_semicolon)
{
   ...
   switch (con->def_kind())
   {
      case CORBA::dk_Operation:
      case CORBA::dk_Attribute:
         // Le operazioni e gli attributi vengono
         // generati da emitPrototypes()
         break;
      case CORBA::dk_Struct:
      {
         CORBA::StructDef_var s
            = CORBA::StructDef_var::_narrow(con);
         emitMIDLStruct(s, emit_semicolon);
         break;
      }
      case CORBA::dk_Union:
      {
         CORBA::UnionDef_var u
            = CORBA::UnionDef_var::_narrow(con);
         emitMIDLUnion(u, emit_semicolon);
         break;
      }
      ...
   };
};
```

Come si può vedere dal codice, gli attributi e le operazioni non vengono trattati in questa funzione, ma è il metodo **emitPrototypes** che se ne occupa.

- **emitForwardDcl** è molto semplice. Si limita a generare le dichiarazioni di tipo *forward* delle interfacce. Nelle dichiarazioni *forward* viene generato solamente il prototipo dell'interfaccia, senza elencare il suo contenuto.
- **emit_sequence** si occupa del *mapping* delle sequenze. Controlla prima di tutto se la sequenza è limitata oppure illimitata e successivamente genera il codice corrispondente (dato che è diverso nei due casi).
- **emit_array** genera il codice per gli *array* e chiama al suo interno la funzione **emit_array_suffix** che si occupa della loro dimensione.
- **emit_midl_type** si occupa della generazione dei tipi. Prima di tutto controlla se il tipo da emettere è di base o è strutturato, quindi chiama di conseguenza i metodi **emit_base_type** oppure **emit_midl_type_name**.

- **emitInterfaceHeader** genera l'intestazione delle interfacce MIDL. Inserisce per *default* i modificatori *object* e *pointer_default(unique)* (per la loro spiegazione vedere il Capitolo 2).
- **insert_guid** inserisce gli UUID per le interfacce e la Type Library COM. Si comporta in modo diverso in base al Sistema Operativo su cui gira il traduttore. Se si tratta di Windows 95/98/NT chiama le funzioni di libreria che creano un nuovo UUID e lo inserisce nel codice; se invece si tratta di un altro Sistema Operativo, genera una sequenza di "X" che dovrà essere sostituite manualmente con gli UUID.
- **emitTypeLib** si occupa della generazione della Type Library. Gestisce anche l'ereditarietà multipla come già abbiamo spiegato in precedenza parlando della funzione **emitMIDLInterface**.
- **CodeGenMIDL** è il costruttore della classe. Gli viene passato come parametro il *database* che contiene le definizioni OMG IDL creato dal parser e non fa altro che inizializzare, in base a questo, le variabili membro *_container* e *_idl_objs*.
- **emit** apre il file di *output* col nome specificato dal parametro di tipo stringa che gli viene passato e chiama la funzione **emitMIDL**.
- **EmitMIDL** è il cuore del traduttore. Con un ciclo *for* scorre tutti gli elementi della variabile *_idl_objs* ed in base al loro tipo chiama le funzioni di generazione del codice corrispondenti. Ecco un suo frammento:

```
void CodeGenMIDL::emitMIDL()
{
   ...
   for (CORBA::Ulong i=0; i < _idl_objs.length(); i++) {
      CORBA::Contained_var c
         = CORBA::Contained::_narrow(_idl_objs[i]->obj);
      if (!CORBA::is_nil(c)) {
         ...
         switch (c->def_kind()) {
         case CORBA::dk_Interface:
         {
            CORBA::InterfaceDef_var in
               = CORBA::InterfaceDef::_narrow(c);
            if (_idl_objs[i]->iface_as_forward)
               emitForwardDcl(in);
            else
               emitMIDLInterface(in);
            break;
         }
         case CORBA::dk_Struct:
         {
            CORBA::StructDef_var s
               = CORBA::StructDef::_narrow(c);
            emitIDLStruct(s);
            break;
         }
         case CORBA::dk_Constant:
         {
```

```
                CORBA::ConstantDef_var co
                   = CORBA::ConstantDef::_narrow(c);
                emitMIDLConstant(co);
                break;
            }
            ...
        }
    }
};
```

Oltre ai metodi elencati, ci sono altre funzioni *helper* che si occupano di estrarre ulteriori informazioni sulla struttura del file OMG IDL da tradurre. Infatti, le classi che descrivono gli oggetti dell'Interface Repository che abbiamo visto in precedenza, non sono sufficienti per fornire tutte le informazioni necessarie per la generazione del codice MIDL.

Ad esempio, nel MIDL i moduli vengono simulati facendo precedere i nomi dei costrutti IDL con una stringa che rappresenta lo *scope* del contesto stesso (vedi il Capitolo 4). Quindi, è stato necessario scrivere la funzione **_midl_absolute_name** che restituisce appunto lo *scope* completo di qualsiasi costrutto.

Le funzioni **emitMIDLException**, **emitMIDLNative**, **emitMIDLValue**, **emitMIDLValueBox** ed **emitValueMember** non generano nessun codice. Sono state incluse nella definizione della classe **CodeGenMIDL** semplicemente per generare un codice d'errore che avvisa l'utente che non c'è nessuna traduzione per questi tipi. Inoltre, in questo modo sarà molto facile implementare queste funzioni quando verrà esteso il linguaggio MIDL o quando sarà disponibile una traduzione ufficiale.

6.4 Uso del traduttore

L'uso del traduttore di Mico è molto semplice. Infatti, supponendo di aver installato la versione 2.2.7 di Mico (o una successiva), per trasformare un file OMG IDL nel corrispondente MIDL basta usare il comando:

```
idl <altre opzioni> --codegen-midl <nome file.idl>
```

Per esempio, per trasformare il file OMG IDL *account.idl* in MIDL si deve usare il comando:

```
idl --no-codegen-c++ --codegen-midl account.idl
```

Il compilatore creerà il file *account.midl* che conterrà la traduzione. L'opzione *--no-codegen-c++* serve per non far generare il codice C++ per l'implementazione.

Nei recenti Sistemi Operativi della Microsoft, cioè Windows 95/98/NT, verranno generati anche i GUID per ogni interfaccia, classe e Type Library; negli altri, invece,

al loro posto verranno generate delle sequenze di X che dovranno essere sostituite manualmente con i GUID. Questo perché le funzioni che li generano sono già incluse nei Sistemi Operativi della Microsoft e non c'è bisogno di procurarsi delle librerie a parte.

Il traduttore mappa correttamente i seguenti costrutti OMG IDL:

- **I tipi di dati di base**. Trasforma il tipo *octet* nel tipo *byte*.
- **Le costanti**. Non è possibile definire costanti di tipo *float* e *double*.
- **Gli enumerati**. Si possono definire al massimo 2^{16} identificatori.
- **Le stringhe**. Vengono tradotte sia le stringhe limitate che quelle illimitate, e sia le stringhe di tipo *string* che quelle di tipo *wstring*.
- **Le strutture**. Traduce anche le strutture ricorsive.
- **Le unioni**. Il *mapping* è effettuato sulle unioni discriminate del MIDL. Traduce anche le unioni ricorsive.
- **Le sequenze**. Vengono tradotte sia le sequenze limitate che quelle illimitate.
- **Gli array**. Se gli array sono di tipo *octet* vengono tradotti in array di tipo *byte*.
- **Le operazioni**. Le operazioni non possono sollevare eccezioni.
- **Le operazioni *oneway***.
- **Gli attributi**. Ci sono delle piccole differenze nel mapping rispetto alla specifica CORBA. Le vedremo successivamente.
- **Le interfacce**. Come abbiamo già visto, rispetto alla specifica CORBA vengono aggiunte delle informazioni sull'ereditarietà multipla nella generazione della **Type Library**.

Ci sono alcune limitazioni sull'OMG IDL che si può tradurre. Prima fra tutte non vengono tradotte le eccezioni. Nel momento in cui è stato scritto il traduttore, Mico aveva una tecnica proprietaria per la gestione delle eccezioni che non era compatibile con la specifica CORBA. Inoltre i diversi Sistemi Operativi su cui è possibile installare (o più precisamente i compilatori che si usano per installare) Mico hanno una diversa gestione delle eccezioni e quindi è molto difficile scrivere del codice universale che funzionerà su ognuno di essi. Per rendersene conto, basta dare un'occhiata alle **mailing list** di Mico, dove molti programmatori hanno segnalato dei **bug** relativi alle eccezioni.

Inoltre, con la specifica 2.2 l'OMG Group ha introdotto all'IDL dei nuovi tipi che per il momento non hanno ancora nessuna traduzione ufficiale (nei capitoli della specifica relativi al mapping non c'è nessun accenno a questi nuovi tipi). Vediamo quali sono:

- **Native**
- **Value**
- **ValueBox**
- **ValueMember**

Non vengono tradotti nemmeno gli **Pseudo Object** che sono dei particolari oggetti che non possono essere invocati tramite la **DII** (**Dynamic Invocation Interface**) e non hanno riferimenti agli oggetti.

6.5 Parte originale

Il traduttore che ho scritto trasforma un sorgente OMG IDL nel corrispondente Microsoft IDL in base alle regole di *mapping* della specifica CORBA. Ci sono solamente due eccezioni che riguardano il nome e i modificatori degli attributi e la generazione della **Type Library** corrispondente. Spieghiamo il tutto con un esempio. Il seguente frammento OMG IDL:

```
// OMG IDL

interface Account
{
   attribute long balance;
   void deposit(in long amount);
   void withdraw(in long amount);
   long read_amount();
};
```

secondo la specifica CORBA dovrebbe essere tradotto in:

```
// MIDL, traduzione secondo la specifica

[
   object,
   uuid(...),
   pointer_default(unique)
]

interface IAccount : IUnknown
{
   HRESULT deposit([in] long amount);
   HRESULT read_amount([out] long *pRetVal);
   HRESULT withdraw([in] long amount);
   HRESULT _get_balance([out] long *balance);
   HRESULT _set_balance([in] long balance);
};
```

ma viene invece tradotto in:

```
// MIDL, traduzione proprietaria

[
   object,
```

```
   uuid(...),
   pointer_default(unique)
]

interface IAccount : IUnknown
{
   HRESULT deposit([in] long amount);
   HRESULT read_amount([out, retval] long *pRetVal);
   HRESULT withdraw([in] long amount);
   [propget] HRESULT balance([out, retval] long *pRetVal);
   [propput] HRESULT balance([in] long NewVal);
};

[
   uuid(...),
   version(1.0),
   helpstring("Account 1.0 Type Library")
]

library AccountLib
{
   importlib("stdole32.tlb");
   importlib("stdole2.tlb");

   [
      uuid(...),
      helpstring("Account class")
   ]

   coclass Account
   {
      [default] interface IAccount;
   };
};
```

La specifica OMG che riguarda il mapping tra CORBA e COM è del Luglio '97, con delle piccole modifiche apportate nel Febbraio '98 e, quindi, non è stata aggiornata in base agli ultimi strumenti di sviluppo forniti dalla Microsoft. Invece, il mio traduttore produce del codice che è compatibile con il Microsoft Visual C++ 6.0.

La principale differenza riguarda il nome degli attributi che, nel primo caso diventa *_get_balance* e *_set_balance*, mentre nell'altro *[propget] balance* e *[propput] balance*.

Il Visual C++ 6.0, quando genera automaticamente lo scheletro del codice che implementa un'interfaccia MIDL, crea una classe **wrapper** che la contiene. Inoltre, il nome degli attributi viene fatto precedere dalle parole *get_* e *set_*. Nel nostro caso avremmo avuto *get__get_balance* e *set__set_balance*. Per eliminare l'antiestetica ripetizione ho scelto di non aggiungere i prefissi *_get_* e *_set_* ai nomi degli attributi.

Lo stesso discorso vale per i modificatori *[propget]* e *[propset]*. Il Visual C++ 6.0 li inserisce sempre, indipendentemente dal fatto che l'interfaccia sia di tipo **dual** (derivata da **IDispatch**) o di tipo **custom** (derivata da **IUnknown**).

L'ultima differenza riguarda il codice per la creazione della **Type Library**, che con il mio traduttore viene generato automaticamente.

6.6 Riconoscimenti ottenuti

Ogni lavoro non serve a niente se è fine a se stesso. Da questo punto di vista, è stata una grossa soddisfazione per me vedere il mio traduttore incluso nella distribuzione CORBA di Mico e leggere il mio nome fra gli autori che avevano contribuito alla sua realizzazione (per maggiori approfondimenti consultare il sito **internet** *http://www.mico.org*). Ma soprattutto è stata una vera gioia ricevere alcune *e-mail* da un tizio della **NASA** (dominio *nasa.gov*) che mi chiedeva delle spiegazioni sul mio traduttore dato che voleva usarlo per un suo progetto.

Capitolo 7

Conclusioni

Cerchiamo adesso di trarre delle conclusioni su questo (lungo) lavoro, mettendo in evidenza soprattutto le difficoltà superate ed i possibili miglioramenti.

7.1 Difficoltà superate

Ogni inizio è sempre difficile, ma il mio lo è stato in modo particolare. Mi riferisco soprattutto alla difficoltà di reperire documenti consistenti sull'argomento.

Se è vero che per quanto riguarda COM la documentazione è abbondante e ricca di esempi, lo stesso non si può dire di CORBA. La fonte principale di informazioni, in questo caso, è stata la specifica CORBA che è molto arida e dispersiva. Per rendersene conto basta dargli un'occhiata. In 1200 pagine circa, gli autori sono riusciti a parlare di tutto senza spiegare niente.

Per quanto riguarda il mapping, ho trovato diversi documenti che ne parlavano, ma ognuno dava la sua implementazione particolare e nessuno motivava quella scelta. Per questo motivo mi sono attenuto alla specifica fornita dall'OMG, tranne in alcuni particolari che abbiamo visto in precedenza.

Passiamo adesso a Mico, croce e delizia di quest'ultimo mio anno. La sua distribuzione consiste di circa 8 MB di sorgente, e non è stato un lavoro semplice cercare di capire il suo funzionamento, dato che dovevo modificarne il codice. In questo caso, una mano consistente mi è stata fornita dal Visual C++ 6.0 grazie soprattutto alle sue funzioni di *browsing* delle classi C++.

Inoltre, c'è da dire che almeno fino alla versione 2.2.5 non c'è stato alcun modo per farlo funzionare su Windows 95/98, ma funzionava solo su Sistemi Operativi di tipo Unix (e quindi anche Linux). Una bella limitazione, considerando che COM funziona solamente sui Sistemi Operativi della Microsoft. Quindi, non avevo nessun modo immediato per testare il codice prodotto dal mio traduttore.

Per fortuna, con la versione 2.2.6, dopo aver ottenuto innumerevoli errori da parte del compilatore e dopo aver applicato tre o quattro *patch* che ho faticosamente cercato nella **mailing list** di Mico, sono riuscito finalmente a farlo funzionare anche su Windows 98. Una vera svolta al mio lavoro, considerando che potevo fare degli esempi concreti di interazione tra COM e CORBA. Da questo punto di vista, è comprensibile la mia gioia e la mia sorpresa quando, compilando la versione 2.2.7 di Mico, non è stato rilevato nessun errore.

Un'ultima cosa da mettere in evidenza. Dopo aver scritto la maggior parte del traduttore e dopo aver raggiunto dei buoni risultati sul mapping usando la versione 2.2.4, è stata rilasciata la tanto attesa (da parte degli utenti di Mico) versione 2.2.5. Per me è stato un vero fulmine a ciel sereno perché erano state ridefinite tutte le strutture dati del *parser* IDL, rendendo il mio codice del tutto inutilizzabile. Quindi, sono stato costretto a riscrivere tutto daccapo.

7.2 Possibili miglioramenti

Passiamo adesso ai miglioramenti che si potrebbero apportare, sia per quanto riguarda la traduzione da OMG IDL a Microsoft IDL, sia per quanto riguarda l'implementazione.

7.2.1 Miglioramenti al traduttore

Nel momento in cui è stato scritto il traduttore, Mico aveva una tecnica proprietaria per la gestione delle eccezioni che non era compatibile con la specifica CORBA. Per questo motivo ho deciso di non implementare la traduzione delle eccezioni. Il problema sembra sia stato risolto a partire dalla versione 2.2.7 di Mico; quindi un possibile miglioramento potrebbe essere il mapping delle eccezioni.

Con la specifica 2.2, l'OMG Group ha introdotto all'IDL dei nuovi tipi che per il momento non hanno nessuna traduzione ufficiale. Questi nuovi tipi sono:

- **Native**
- **Value**
- **ValueBox**
- **ValueMember**

Si potrebbe fornire un metodo proprietario per mappare questi tipi, anche se un discorso a parte meritano gli ultimi tre che riguardano gli **Object by Value**.

In un'applicazione distribuita, ci sono due possibili metodi per ottenere l'accesso ad un oggetto situato in un altro processo. Il primo è conosciuto come **Object by Reference**, il secondo come **Object by Value**.

Supponiamo che un processo **A** passi un riferimento ad un suo oggetto al processo **B**. Nel primo caso, quando **B** invoca un metodo sull'oggetto remoto, il metodo è eseguito su **A**, perché è questo processo che possiede l'oggetto, l'oggetto esiste in memoria nello spazio di indirizzamento di **A**. Il processo **B** ha solo la visibilità dell'oggetto tramite il suo riferimento. Nel secondo caso, invece, quando si passa il riferimento, lo stato corrente dell'oggetto (come ad esempio il valore attuale delle sue variabili membro) è passato al processo **B**, tipicamente tramite un processo di *serializzazione*. In questo caso, quando un metodo è invocato da **B**, viene eseguito su **B** stesso invece che su **A** dove risiede l'oggetto originale. Quest'ultimo, quindi, non subisce nessuna modifica del suo stato proprio perché è stato passato per valore.

Generalmente, spetta al programmatore fornire il codice per la *serializzazione* e la *deserializzazione* degli oggetti. Riassumendo, quando un oggetto è passato per valore (**Object by Value**), lo stato dell'oggetto è copiato e passato alla sua destinazione dove viene istanziata una nuova copia dell'oggetto. Quando verranno invocati i suoi metodi, verranno processati sulla copia e non sull'oggetto originale.

Attualmente CORBA supporta sia il passaggio **by Reference** che **by Value**, mentre COM supporta solamente il primo. Si potrebbe allora cercare un metodo per simulare su COM il passaggio **by Value**, anche se non è una cosa semplice da fare perché presuppone un minimo di implementazione anche da parte dell'architettura.

7.2.2 Miglioramenti nell'implementazione

Il **tool** che ho scritto, che è stato inserito nella distribuzione Mico, si limita a tradurre dall'OMG IDL al Microsoft IDL, e non genera nessun codice per l'implementazione della **View COM**. Quindi, si potrebbero aggiungere alcune opzioni da riga di comando per fargli generare anche questa.

Ovviamente l'implementazione cambia in base al tipo di server che si vuole creare, e cioè **InProc**, **OutofProc** e **Remote**; ma anche questo potrebbe essere specificato da riga di comando, ottenendo così tre possibili tipi di implementazioni diverse.

In alternativa si potrebbe scegliere di far generare automaticamente solo la parte di implementazione comune a tutti e tre i tipi di server, lasciando al programmatore il compito di aggiungere il codice rimanente.

Anche l'implementazione può essere fatta in modi diversi. Ad esempio si potrebbe usare l'**ATL**, in modo tale da semplificare al massimo il codice da generare, oppure si potrebbe usare solamente il C++ puro, per non dipendere da nessuna libreria in particolare.